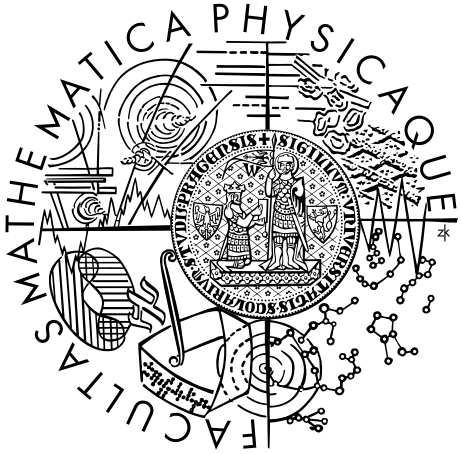
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Ondřej Dvořák

Projectional editor for domain-specific languages

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Michal Malohlava

Study programme: Informatics

Specialization: Software systems

Prague 2013

First of all, I would like to express a great appreciation to my supervisor Michal Malohlava for his valuable input during the writing of this thesis. His thoughts and tremendous support had a major impact on this work. I would also like to thank to my family and friends for cheering me on through my entire studies.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague on April 8, 2013

Ondřej Dvořák

Název práce: Projektivní editor doménově-specifických jazyků

Autor: Ondřej Dvořák

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Michal Malohlava

Abstrakt: *Programování je proces návrhu algoritmu, psaní, testování a ladění zdrojového kódu. Ten se neobejde bez existence nejrůznějších programátorských nástrojů, jako je například integrované vývojové prostředí (IDE) umožňující správu většiny univerzálních jazyků s textovou reprezentací. Vzhledem ke vzrůstající popularitě doménově-specifických jazyků (DSL) je nezbytné, aby IDE podporovalo i je. DSL jsou ovšem reprezentovány nejen textovou formou, ale i grafickou a případně jejich kombinací. Jedním z nadějných přístupů je nová metoda zvaná projektivní editace. Jejím úkolem je umožnit různé způsoby zobrazení a manipulace se zdrojovým kódem. Tato myšlenka je většinou realizována projektivním editorem.*

Tato práce se zabývá návrhem a experimentální implementací projektivního editoru pro doménově-specifické jazyky. Analyzuje možné přístupy k projektivní editaci a navrhuje jejich vhodnou aplikaci v Microsoft Visual Studiu. Přináší univerzální implementaci projektivního editoru integrovaného ve Visual Studiu i v samostatné aplikaci. Tento je navíc rozšiřitelný o nové vizuální formy daného DSL.

Klíčová slova: projektivní editor, workbench, ide, dsl, programování

Title: Projectional editor for domain-specific languages

Author: Ondřej Dvořák

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Michal Malohlava

Abstract: *Programming is a craft requiring a good tooling. One of tools selected as crucial for software development is an integrated development environment (IDE) that allows to maintain most of the general-purpose languages. Domain-specific languages grow in a popularity last years, thus it is necessary to support them by IDE as well. Not just a textual or graphical form of DSL sources is suitable for their maintenance, so does the combination of them. One of the promising approaches is represented by a novel method called a projectional editing. Its objective is to show different visualization forms of program source code, combine and manipulate with them at one place. The thought is typically realized by a projectional editor.*

In this thesis we design a projectional editor for domain-specific languages and provide its experimental implementation. It analyzes potential approaches to a projectional editing and designs their suitable realization in Microsoft Visual Studio. It provides a universal implementation of a projectional editor on the top of Visual Studio as well as on the top of a standalone application. Moreover, it supports its further extension with new visualization forms for a given DSL.

Keywords: projectional editor, workbench, ide, dsl, source code, programming

Contents

Chapter 1

Introduction and Motivation

A software is directly or indirectly affecting our everyday life. Hence a software has to meet a demand for high-quality, performance, stability, and a user-friendliness in order to gain the required level of popularity. To maintain such a popularity level, the software development should be flexible and adaptable enough to the sensitive users' needs. Therefore, the high development effectiveness is a must. Unfortunately none of the mainstream languages like C# or Java enable such an effective approach because of the code complexity. Especially in bigger development teams where a modification or a bug fixing of a foreign code has to be usually done by an uninitiated programmer, the code has to be kept as clear as possible, which can be sometimes quite difficult because of its complexity. Nevertheless, not only the complexity of the code is a bottleneck. The effectiveness of a communication between developers and domain experts has almost the same impact to the development as a coding process itself. Therefore, the ability of domain experts to comprehend the code or at least its main context would accelerate the time to find a bug in the core domain design of an application. Here the *domain-specific languages* [1] come into play.

A *domain-specific language* (DSL) is worth to be used, if it facilitates a solution of a particular problem and it can express it more clearly than any other existing language. In contrast to the common languages like C# or Java mentioned above, the DSL is not aimed at solving any software problem, but its main goal is to ease and clear the solution of a limited problem domain. Due to the simplicity of DSL, several visual and textual forms can be used for its presentation. These forms are known as DSL *projections* ([2],[3]) (see the *Figure 1.1*). Since some of the projections do not require any deeper technical programming knowledge and can be easily understood by a non-programmer, the cooperation between programmers and domain experts can be established via distinct DSL projections operating over the same problem domain. Editor supporting multiple code projections is so called *projectional editor*.

The projectional editor can be either a standalone application or a part of

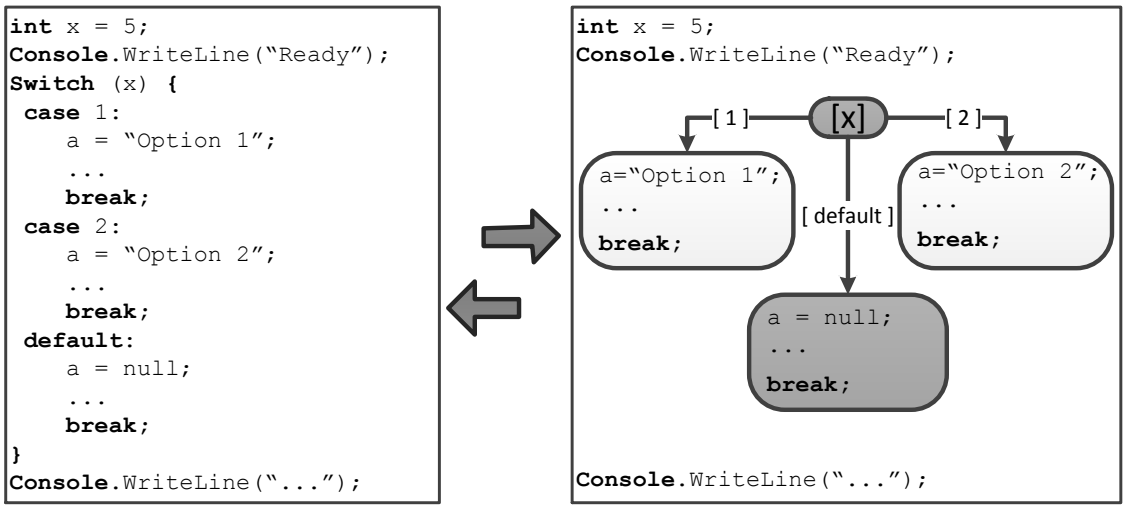


Figure 1.1: A code snippet represented by two projections, a textual on the left and a graphical on the right

an IDE. Nevertheless, this is not the only area where the projectional editor holds its place. From a general point of view, such an editor is not just a tool serving developers to edit the DSL code. A common GUI form (e.g. tax form) can be also considered as a projectional editor. Its only projection maps the object being edited by the form to the form itself. Thus the usage of a projectional editor may be found among the usual application users as well as among the application developers.

In spite of the requirements of the IDE and the projectional editor-like client applications can be slightly different, the most fundamental component performing the projection itself can be reused. Therefore, a design and a development of such a reusable component are well-founded, and should be considered as a key part of the projectional editor.

1.1 Goals

The overall goal of this thesis is to analyze possible approaches to the projectional editing, design a universal projectional editor and implement it on a top of the selected IDE. It includes a description of its limitations and capabilities as well as a demonstration of its usage on an example. Since the choice of an IDE directly influences the whole analysis (see *Chapter 4*) and implementation (see *Chapter 5*), the IDE must be selected before.

Because there are tons of various programming languages and several IDE for their maintenance, we have to concentrate only to those we consider important. There is no need to conceal that this choice is driven by experiences and an affection for some of them and thus it is narrowed down to Microsoft Visual Studio (MSVS)

related to .NET [1] and Eclipse dedicated to Java [2].

Nowadays, none of them is provided with a widespread projectional editor. Thus from this perspective, both the candidates seem to be equivalent. However, there already exist several prototype solutions in Eclipse as well as Java tools operating outside this IDE. A pure projectional editor dedicated for .NET and MSVS does not exist at all.

Since the projectional editor is missing in Microsoft Visual Studio, it seems to be the most suitable target platform for our solution. With regard to personal preferences, C# becomes a programming language used for the implementation. Thus we will not in any way question our choice further in this work and all the technologies and tools not compatible with MSVS and C# will be omitted within this thesis.

1.2 Structure of the Text

Within the following chapters the problematic of DSL projectional editors is going to be discussed. For the deeper understanding of the thesis, the domain knowledge and various technical skills are required. Therefore, the next *Chapter 2* introduces DSL, projections, projectional editors, technologies, and software development patterns used. The *Chapter 3* revisits the goals of the thesis. Their deeper analysis is contained in the *Chapter 4*. The design and architecture of the implemented solution are shown in the *Chapter 5*, whereas the usage of the given solution is demonstrated in a subsequent *Chapter 6*. The related work is discussed in the *Chapter 7* and the overall evaluation of the proposed solution can be found in the *Chapter 8*. The last *Chapter 9* concludes the thesis and gives possible ideas for a future work.

Needless to emphasize that the revisited goals are initially decomposed into various sub-problems (decomposition steps). They are analyzed successively in the *Chapter 4*, while each section corresponds with one section in the *Chapter 5*. These are organized in a completely same order. See the figure below.

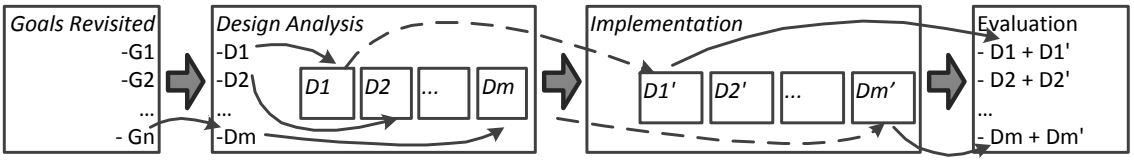


Figure 1.2: Structure of the thesis

[1] Microsoft Visual Studio IDE (see *Section 2.4*) is tightly coupled with .NET (see the *Subsubsection 2.4.1.1*) languages (C#, Visual Basic) and it became one of the most popular IDE for them.

[2] Eclipse is an IDE for Java, programming language created by Sun Microsystems.

Chapter 2

Domain Introduction

The aim of this chapter is to introduce several important terms and technologies and provide us with a fundamental knowledge that is crucial for the broad understanding of this thesis. Knowledge acquired after reading this chapter should be sufficient for the theoretical part of this work - analysis (see the *Chapter 4 Design Analysis*), as well as its practical part - implementation (see the *Chapter 5 DSLPed Implementation*).

2.1 Programming Languages

Programming languages can be divided into general-purpose languages (GPL) and domain-specific languages (DSL). Since this work targets DSL only, they worth to be introduced properly. To distinguish them from GPL, a brief description of GPL must be done before. Both the language categories are discussed in this section.

2.1.1 General-purpose Programming Languages

Programming languages like Java, C# or C++ can be considered as general-purpose programming languages. A software written with their use may belong to a wide variety of application domains. Not to make a confusion, an application domain is just a world where the application lives in. For instance, if we are about to develop a *traffic simulator*, the application domain would be *traffic simulators*.

Since the software written using GPL targets many fields of work and infinitely many situations, the syntax of such a language does not contain any language constructs[1] intended to ease the coding of a program that comes from a specific application domain. The size of the target application domain is in fact one of the main differences from the domain-specific languages described below in a detail.

[1] *Language construct* refers to a syntactic structure or a set of structures of a language that represents an allowable part of a program in accordance with the language rules.

On the other hand, a *problem domain*, sometimes called a problem space is just a term that refers to the information leading to describe a problem and constraint its solution under the given application domain.

2.1.2 Domain-specific Languages

There exist many various definitions of DSL. A book *Domain Specific Development with Visual Studio DSL Tools*[11] says:

> *"Domain-specific Language is a custom language that targets a small problem domain, which it describes and validates in terms native to the domain."*

In other words, DSL is a programming language aimed at solving only particular problems we have. Due to the small problem domain which such a problem comes from, the solution can be expressed in an elegant and smart way. Moreover, potential complexity of the same problem solution with the use of a general-purpose programming language can be easily hidden thanks to the possible simple and clear visual forms of the DSL code.

For instance, HyperText Markup Language (HTML) can be considered as DSL. If the structure, layout and behavior of the web page would be described by GPL, the code complexity would increase significantly in comparison with a HTML code. The written form of HTML consists of XML[2] tags and their attributes which define how the web page looks like in a web browser. Since all the properties of a web page come from the limited domain, domains of tags and attributes are limited as well. For example, the text on a web page can be aligned either left, right or center. The alignment itself is defined by a corresponding HTML attribute *align* and the only values to be assigned to this attribute are *left*, *right* or *center*. Thus the domain of an attribute *align* is as limited as the domain of possible text alignments is. With respect to the DSL definition, the attribute *align* is a solution of a specific problem - *how to align a text on a web page*, therefore no general construct *align* exists in GPL. Such a solution coded using GPL would be apparently much more complex and complicated. Because of several similar problems like *text coloring*, *text size* or *text style* the developer is faced regularly during the web page programming, the DSL is relevant to be applied to the web page development.

On the other hand, the creation of a DSL may not make a sense sometimes, despite the usage of a designed language would ease our problem solution. If the time for such a DSL implementation would significantly exceeded the time for the solution using GPL and would be impossible to apply our DSL on similar problems,

[2] eXtensible Markup Language

then DSL seems to be an overkill. The type of a given problem should repeat sufficiently often in order to design a DSL for its solution.

2.1.3 Syntax and Grammar of a Language

In general, the syntax is a way how the given programming language can be combined in order to create a well-formed program. It defines the relations between constructs of a language, together with a description of how the various expressions that make up a legal strings look like. For instance, if there is a variable or a method contained in the language, the syntax defines how its name may look like.

The syntax itself is usually described by a kind of a formal grammar, which refers to a way of how the symbols of a language can be formed into valid language phrases. One of the possible notations to describe a syntax is using a meta-syntax called Backus Normal Form (BNF). Since the BNF, as well as the theory of languages exceed research within this thesis, see details in [13].

The syntax of a DSL may be described by XML schema. XML Schema Definition (XSD) is a schema describing the structure of a XML document, its elements, attributes and their relations. Beside that, it allows to validate given XML document against the specific XSD. The syntax it describes is in fact just an *abstract syntax*. It is independent of any encoding and representation of a language. It can be considered as a data model. An abstract syntax structure of source code can be expressed by a tree representation called *Abstract Syntax Tree* (AST). The nodes of a tree denote to the constructs of language. A specific machine representation must be mapped (see [4]) from the abstract syntax using so called concrete syntax, the corresponding derivation tree is called *concrete syntax tree*, or simply parse tree.

However, if we restricted only to those DSL whose structure can be kept in a XML file, the XSD could be considered as their grammar. Moreover, their syntax could be validated against the XSD file.

2.1.4 DSL Models and Their Instances

A conceptual model related to a particular problem is well-known as a *domain model*. All the constraints governing the problem domain are described by a domain model in a form of entities, attributes, roles and relations among them.

The design-time representation of any DSL is a domain model created with respect to the problem, which the language should be applied to. Since the term domain model is not entirely accurate expression for a model of a language, from now we will call a domain model of a language just a *model*.

Model of DSL is in fact another way to describe a syntax of a language. Needless to say that it describes only the abstract syntax of a language. Such a model

can be easily managed and presented in a form of entity-relationship diagrams (ER), where the entities stand for constructs of a language. If there already exists a XSD file describing the syntax of a language, we can create a model of DSL out of it. The reverse transformation of created model back to the XSD is definitely the most fundamental advantage when designing the DSL. We can simply create a model in a form of ER, which is in fact a natural way to do so, and transform it back to a XSD file.

The class representing the DSL can be generated out of its XSD description, regardless the programming language used for its implementation. The runtime instance of a generated class is known as a *model instance* and refers to a source code written in the language represented by a model.

2.2 Projectional Editors

Projectional editor is the main subject of this work. Not only this thesis, but the whole concept of projectional editors revolves around a term projection. Thus a term projection is too essential for us, to omit attempts to understand it. A substantial presence while reading the following sections is therefore desirable.

2.2.1 Projections

Projection is a term of several definitions in different fields of science. In this thesis projection is meant by a form of a presentation of the native source code.

The code is usually presented in a textual form which is in fact a kind of a projection to the representation that can be processed by a compiler. In order to facilitate and streamline the code management, the tools for a code formatting and syntax highlighting can be used. These are known as pretty-printers, applications that apply several formatting conventions to a plain text or a source code. The formatted and syntactically highlighted code within any text editor is just another clearer form of a code presentation, thus it is a projection as well. This projection is nowadays more or less a natural way to visualize the source code. Nevertheless, many other projections can be found.

Let us consider a programming language describing a workflow. Workflow consists of a sequence of concatenated steps of a given process. Regardless the specific syntax of the language, at least two different projections can be found, the native text projection and a graphic projection of nodes concatenated according to the given workflow. Apparently more than one form of a presentation can be used when the code of such a language is managed.

Since the projection is really just a presentation of a given code, all the changes

done within the projection are in fact being done directly in the code itself. Therefore all the changes done within one projection should be reflected automatically to another projection of the same code. Such a behavior is tightly coupled with a term *projectional editor* which is a subject of the next section.

However, not every language is suitable enough to be presented in other than textual or pretty-printed form. That happens mainly because of the complexity of certain languages. GPL are an example of languages where it is almost impossible to find another reasonable projection than the simple textual one. Nevertheless, projections of partial GPL constructs can be worthwhile. For instance, a two-dimensional array can be presented in a form of a grid, notwithstanding the rest of a code remains presented like a text. This kind of projections is nevertheless out of scope of this document. DSL are much more suitable to be presented in various forms because of their relative simplicity, therefore the projection is always understood as a form of presentation of any DSL within a content of this thesis.

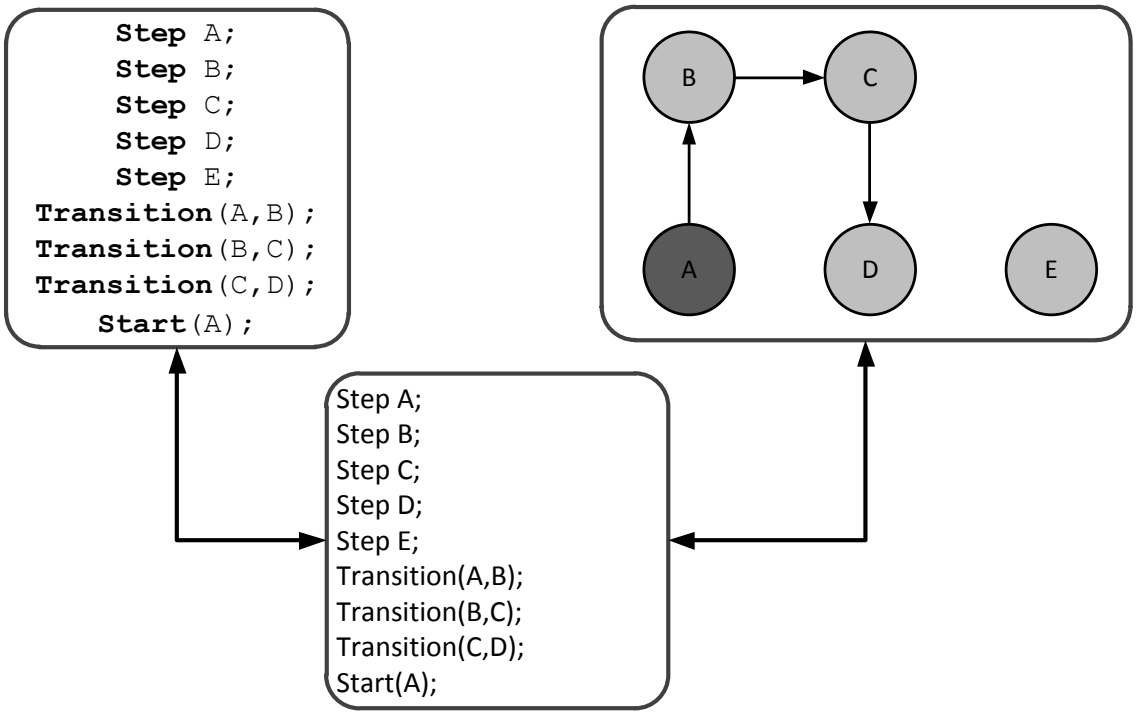


Figure 2.1: A simple view of DSL projections. Each of them presents the same source code, once as a highlighted text, once in a form of plain text and finally in a graphical form.

2.2.2 Projectional Editors

Projectional editor, also known as structured editor, is an editor intended to manipulate the code of a given DSL in various projections. It is designed to be either a standalone application or a part of the IDE. Editor should support several types

of projections, from a native textual one to a sophisticated graphical projection according to the given DSL.

In general, if a document content is provided with a clear and well-defined structure, the projectional editor might be used to handle this content. On the other way around, a plain text usually does not have any deeper thought structure, thus a projectional editor would be powerless to handle such a content in other than a text projection. Since a plain text handling has mostly different requirements than the ones that a projectional editor can satisfy, the maintenance of plain text files is usually done with the use of common *text editors*.

2.3 Design Patterns

Several concepts in this thesis are driven by various design patterns. The most interesting ones worth to be introduced briefly and these are described in this section.

Design pattern is a solution of a common software design problem that frequently occurs. In fact, it is not directly a solution of a specific problem, but only a reusable template that fits to a specific problem. Design patterns started to grow in popularity after the book *Design Patterns: Elements of Reusable Object-Oriented Software*[5] was published by so called *Gang of Four* (GOF). The book is a set of design patterns of common software problems.

2.3.1 Model View Controller

Model View Controller (MVC) is aged design pattern originated in a Smalltalk system in 1970. It was discussed in the GOF book [5], as a pattern gaining the popularity in applications, where it is reasonable to separate the data representation and user's interaction with them. The application is divided into three layers under this pattern. A *Model*, which could be considered as a business logic of the application, a presentation layer called *View* and a *Controller* that maintains the interaction between the user and GUI. In the next Section MVVM pattern is discussed. Since MVC is the basis for MVVM, it seems reasonable to have at least the fundamental notion about it.

2.3.2 Model View ViewModel

Model View ViewModel (MVVM) is MVC-based design pattern introduced by Microsoft. It cleanly separates a graphical user interface (GUI) from the business logic of the application and thus targets to various graphical subsystems. Such a separation facilitates the maintenance of the application and its testability. The application that follows MVVM pattern is organized in three separated layers. A

Model layer representing the business logic and data to be presented, a *ViewModel* layer encapsulating the logic of presentation and *View* layer that corresponds with the presentation part of the application. Since it is a kind of a fresh pattern, it was not covered in the GOF bestseller [5]. Nevertheless it is discussed in a context of Microsoft technologies in a book [12].

2.3.3 Factory Pattern

Factory Pattern has arisen to deal with the problem of objects creation. It in fact serves to create an object in a similar way as the *new* keyword in several programming languages does. Nevertheless, if the creation of such an object should be centralized, it is reasonable to delegate this function to a central authority - *Factory*. Although in this thesis, the factory pattern is not used in its pure form, several components in the implementation part are based on it. Therefore, it is wise to be aware of its basic concept.

2.4 Microsoft Visual Studio

Microsoft Visual Studio (MSVS) [6] is an integrated development environment (IDE) designed and created by Microsoft. Beside the console applications, web applications and several others, it supports the development of graphical applications. Although it integrates most of common IDE components and features like a code editor with IntelliSense, refactoring, debugger, and various designers, it allows the developer to extend certain of them or even integrate completely new components in it. Since a goal of this thesis is a projectional editor integrated in MSVS, it tends to a creation of new graphical components to MSVS and thus the topic of creation MSVS components becomes a point of interest for us.

To develop new components and integrate them successfully to MSVS, we have to introduce runtime environments in which they operate, we have to propose the graphical subsystems they support and present frameworks by which they can be extended with new information (mainly user-defined domain-specific languages and their projections). Finally, we have to pack new components into redistributable units, easy installable to MSVS, thus we have to bring up all possible packaging tools.

Issues noted in the last paragraph are discussed step by step below in this section. It is more or less an overview of all available technologies, their suitability will be evaluated in the *Chapter 4*.

2.4.1 Runtime Environments

A software platform that is reusable and may be used for the development of applications is so called *software framework*. A *virtual machine* refers to a software that is able to execute programs like the physical machine does. In our thesis, we are dealing with *Common Language Runtime* (CLR), a virtual machine that may execute programs written in a certain .NET language and is coincidentally supported by MSVS. It is a part of the .NET Framework described in the following subsection.

2.4.1.1 .NET Framework

.NET framework [10] is a runtime environment intended for a development and deployment of rich applications that can be executed with the use of CLR. Whenever the .NET framework is installed, the application targeted the framework can be executed despite the hardware and operating system configuration of a given machine. Cross-language compatibility of .NET framework languages like C# or Visual Basic (VB) is supported, what in fact means the .NET components can interact with each other regardless the language which has been used for their development. Due to this benefit all of the code snippets and examples in this thesis can be demonstrated in a C# code, since the VB code would be mostly equivalent.

2.4.2 Graphical Subsystems

In a context of this thesis, by a *graphical subsystem* is meant a tool that is responsible for displaying the output of an application to the screen. We are interested in the ones backboned by .NET framework and those that MSVS components can use to display their output.

2.4.2.1 Windows Presentation Foundation

Windows Presentation Foundation (WPF) [16] is a graphical subsystem that can be used to render graphical user interfaces in Windows graphical applications. It has been released as a part of the .NET framework 3.0. Due to its programming concepts which enable the separation of a user interface and the business logic, the MVVM design pattern can be applied on WPF applications naturally. Whereas the user interface is designed by a XML-based XAML code, the business logic is implemented using a .NET language, usually C# or Visual Basic.

2.4.2.2 Windows Forms

Windows Forms (WinForms) [18] is a graphical application programming interface (API), which supports the direct access to the native Microsoft Windows interface.

It is an aged tool that provides a similar functionality as the WPF does. However it was not replaced by the WPF and it exists in parallel to it.

2.4.3 .NET Extensibility Frameworks

The injection of various information into .NET applications (or components) usually happens via a certain .NET extensibility framework. Since MSVS hosts various .NET components, we are interested in extensibility tools dedicated to .NET. These are briefly described below.

2.4.3.1 Managed Extensibility Framework

Managed Extensibility Framework (MEF) [8] is a tool for a design and creating of lightweight, extensible .NET applications and components. An extension may be easily discovered in runtime and afterwards used as a common .NET object, while in fact no configuration is required to discover the extension. They can be provided with metadata attributes and thus postpone the need of object instantiation. This can be worthwhile, when the object instantiation is an expensive operation. One of the biggest benefits of MEF is that it allows to reuse extensions within components of one application. Moreover, these can be even reused across various applications as well. Due to such a reuse of components, some see MEF as an IoC container[3]. However this point of view is controversial and is mentioned just to make an impression about what is MEF.

Starting the MSVS of version 2010, most of the features of an internal code editor can be extended via *MEF Components*. MEF can be used for the enhancement of appearance and behavior of an internal editor only, so basically stuff like margins, scrollbars, adornments or IntelliSense can be contributed by MEF. Although MSVS does not provide the extension of another components via MEF, similar scenario can be probably applied to new components that require further extensions by MEF.

2.4.3.2 Managed Add-In Framework

Managed Add-In Framework (MAF) [9] is a Microsoft framework designed to manage extensions of applications. Its focus is a big higher-level than a focus of MEF. It is based on an add-in model and targets an independent versioning, discoverability and activation of an add-in. It supports several levels of isolation between the hosting application and an add-in itself.

[3]Inversion of Control (IoC), a process of a dependency injection.

2.4.4 Package Distribution

To provide developed components as units installable to MSVS, we have to choose a suitable form for their distribution. These are usually a kind of packages that encapsulate new component appropriately and that can be integrated into MSVS on a common basis.

2.4.4.1 Visual Studio Integration Package

Not all the MSVS 2010 parts can be extended via MEF. For instance, additional UI elements, services, editors or designers are just too large and *Visual Studio Integration Package* (VSPackage) [7] is a more suitable form for their integration. However, Microsoft Visual Studio SDK (MSVS SDK) for a targeted version of MSVS is required for the development of VSPackages.

2.4.4.2 Visual Studio Extension

Visual Studio Extension (VSIX) is a format that can be used for the packaging of various MSVS extensions like VSPackages, project templates, item templates, MEF components, assemblies, toolboxes and custom types. VSIX deployment requires MSVS SDK to be installed and if the VSIX package is uploaded into Visual Studio Gallery website, the extensions can be easily checked out and installed online.

Chapter 3

Goals revisited

Domain-specific languages have been presented in the previous chapter together with their benefits for a programming. We found out them as a suitable category of languages that can be treated by projectional editors, whereas the GPL were too clumsy for them. The .NET Framework itself has been introduced as well as various technologies operating in it. Thus the portfolio of knowledge needed to understand the given problematic deeply should be sufficient now. The revision of all the goals we have with respect to this domain knowledge is the exact topic of this chapter.

The general objective of this thesis is to analyze possible approaches to a projectional editing and to design and implement a projectional editor for *domain-specific* languages. To clarify it, a framework for a definition of DSL presentable in projections is not a focus of this thesis. We are interested in their projections in editor only. The editor should permit the developer to edit and show DSL source code in various visualization forms and should be extendable of new ones for a given DSL. We will therefore establish an overall goals to be achieved based on this objective. The reasons which have led us to choose a specific technology for our solution will be analyzed deeply in the next *Chapter 4*.

We want to propose a solution, which will enable to show a source code of any DSL in various defined projections. We want to present a well-defined and unique way how these DSL and their projections can be defined and how the projectional editor can be extended of them.

The second important goal is to realize such a solution as an extension of a chosen IDE with the use of a specific programming language. In our work we decided to integrate the solution into Microsoft Visual Studio IDE using the C# programming language. Therefore we have to find a suitable C# compatible approach for the definition of DSL grammars and their projections. Beside that we have to provide an implementation of a projectional editor that can be integrated into the Visual Studio and extended of new DSL and their projections using a convenient technology.

Thus the main goals of this thesis are:

- G0: analyze possible approaches of a projectional editing
- G1: propose an architecture of MSVS projectional editor for DSL
- G2: the editor should be able to handle new DSL
- G3: the editor should support custom projections
- G4: develop the proposed editor and integrate it into MSVS
  - G4a: allow hops between projections in dependence on interactions with a user (*events*[1])
  - G4b: allow switching between projections
- G5: demonstrate the usage of the implemented solution on a specific DSL

[1]Event is meant by a specific user action within a projection, like mouse down, button click, etc.

Chapter 4

Design Analysis

Within the previous chapter the goals of this master thesis were revisited. In order to satisfy them, their deeper analysis with respect to the available technologies must be done before. The satisfaction of all the stated goals is an overall problem to solve. The decomposition of the problem together with basic ideas which led us to choose a specific technology for our solution can be found in this chapter.

The design and architecture of the MSVS projectional editor will emerge naturally from the analysis of possible projectional editing approaches, thus the goal G0 and G1 will be satisfied then. Fundamental design components must be introduced together with their abilities and characteristics as well as their possible integration within the MSVS. To achieve the goals G2 and G3 requiring the extensibility of projectional editors with new DSL and custom projections, a deeper analysis of available technologies and tools must be done before. The separate *Chapter 5* deals with technical problems associated with the implementation of the solution, thus the goal G4, which is purely technical, will be satisfied together with its sub-goals within that chapter. However a sub-goal G4a deserves to be slightly analyzed before it is implemented, thus it is discussed in the *Section 4.5* at first. Goal G5 is a kind of a tutorial how the make a new DSL extension to the created projectional editor. This will be discussed in the *Chapter 6*.

4.1 Problem Decomposition

As soon as we proceed with the problem decomposition we will result in the definition of sub-problems (decision steps), which we will discuss separately within the subsequent sections. We will try to decompose the problem step by step with respect to the goals we have already revisited.

Editor is a tool that allows us to manage a certain programming language code. To achieve that, it must know how the languages looks like, what are its keywords

and rules for how to organize them in order to create a valid language constructs. This is usually done by a kind of a formal grammar, which describes exact way the keywords and rules of the language. On the other hand, nothing prevents us from describing it verbally or using a meta-model of the language.

≫ ***D1: The structure of a language must be described*** *(see analysis Section 4.2, implementation Section 5.3)*

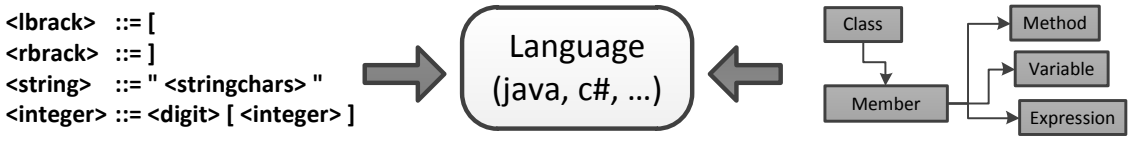


Figure 4.1: Two possible ways of a language description. A grammar on the left, a meta-model on the right.

Based on the knowledge of a language structure, different forms of its visualization must be proposed. These are in fact the ways how the language constructs are displayed to the user. For example, they describe whether a plain text or a flow diagram is used for the visualization of a switch statement, or if a specific keyword is separated by underscores or spaces when it is presented to the user. The visualization forms of these constructs are the *projections* and we need to find a suitable way to define them. In addition to that, we should keep in mind, there are usually several different language constructs and if we want to visualize any, at least one projection must exist for it.

≫ ***D2: A suitable way to define projections of language constructs must be designed*** *(see analysis Section 4.3, implementation Section 5.4)*

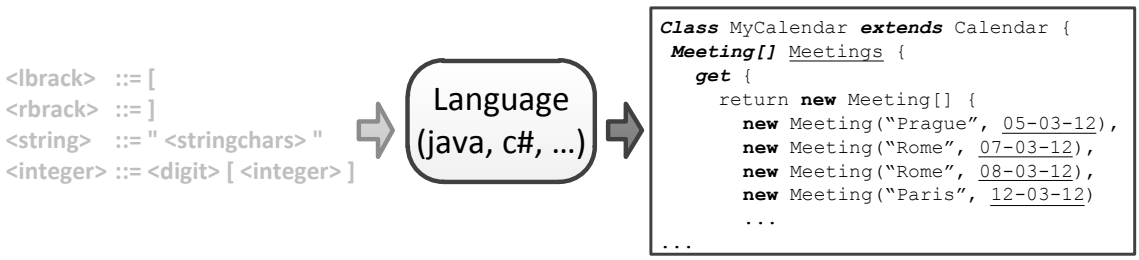


Figure 4.2: An idea of a projection that visualizes a language appropriately. A language in the middle is presented in a projection on the right.

However, there is not always only one projection of a specific language construct. These must be tied with a construct conveniently. In fact, a language construct must be enriched with a knowledge of its own projections. An instance of such an enhanced construct is called *projective object* since it is able to present itself in various defined projections. Therefore, we must find a way to create a projective object out of a specific language construct.

≫ ***D3: Language construct must be enhanced of projections and thus brings a projective object*** *(see analysis Section 4.4, implementation Section 5.5)*

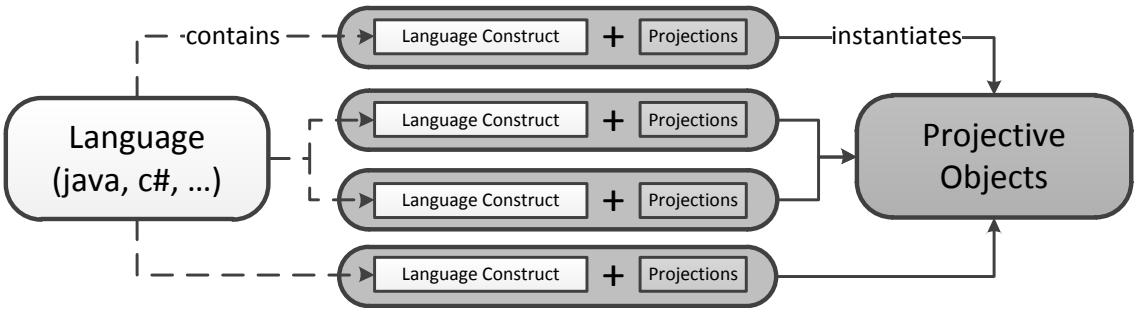


Figure 4.3: A language source code is built from various language constructs. These must be joined with their projections and instantiated as projective objects.

With respect to the goal G4a, event-driven hops between projections should be possible. It means there exists a kind of a workflow among projections. This workflow is driven by certain events and we have to describe the workflow appropriately. Since these projections appear in dependence on events, let us call them *event-driven projections*.

≫ ***D4: We have to describe and keep a workflow between event-driven projections*** *(see analysis Section 4.5, implementation Section 5.6)*

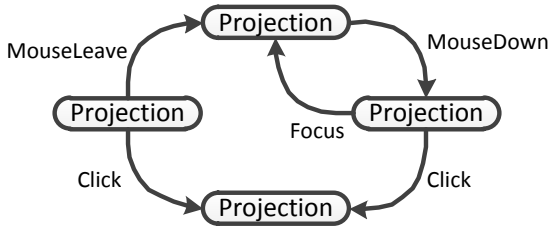


Figure 4.4: Different projections define circumstances under which they hop to another projection. The circumstance is given by an event (Click, Focus, etc.).

There are usually more usable projections of a specific language construct and it would be appropriate to allow switching among them, even without raising an event. This is exactly the idea of projectional editors. The projectional editor is provided with several different projections and the user should be able to switch among them, while changes made within one projection are automatically reflected to another. In general, the visual control that manages a bunch of projections is a fundamental component of the whole projectional editor.

≫ ***D5: A general component for managing the projections of one specific construct is necessary. Let us call it projective component*** *(see analysis Section 4.6, implementation Section 5.7)*

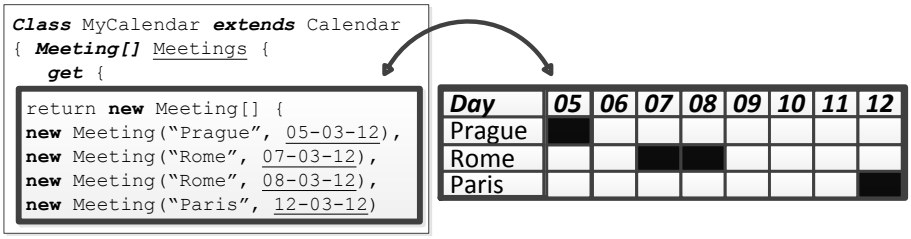


Figure 4.5: Projective component that manages two different projections, a textual projection on the left and a grid projection on the right.

An instance of a language is a piece of code written in it. It usually contains many constructs of the same type. For example, an instance of the Java language contains several methods. No matter what the names or parameters are, all of them have the same type - *method*. These are provided with the same set of projections. If we wanted to change the projections of all at once, we would need to call a function operating over all the projective components that manage them. This is a kind of a global function that does not belong to the projective component itself, but it belongs to a parent component instead. In this case, such a responsible component is represented by an editor, which takes care of all the underlying projective components and provides features applicable to them. This is so called *projectional editor*. The question is, how to implement such an editor composed of projective components and how to establish the intercommunication between them and the editor itself.

≫ ***D6: The composition of the projective components must be performed to build an editor*** *(see analysis Section 4.7, implementation Section 5.8)*

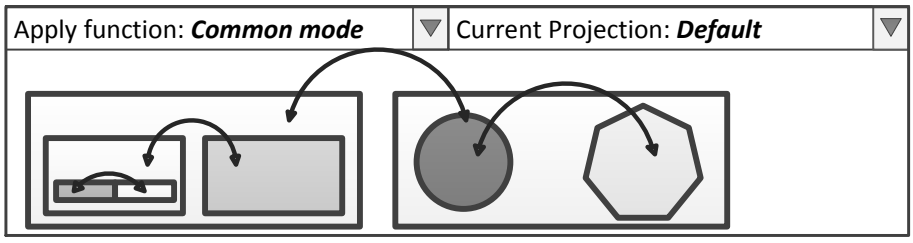


Figure 4.6: Projectional editor composed of nested projective components.

Extensibility, another problem we have to face. There is usually not just one domain-specific language we want to support by the editor. The use case is that a developer will be provided with a grammar of the language and accurate demands on its visual appearance. Definitions of the projections will be created based on this information. The developer will combine projections with grammar and injects them to a projectional editor. Thus we have to analyze the projectional editor from the perspective of its extensibility with new domain-specific languages.

≫ ***D7: The design must cover an editor extensibility with new languages and their projections*** *(see analysis Section 4.8, implementation Section 5.9)*

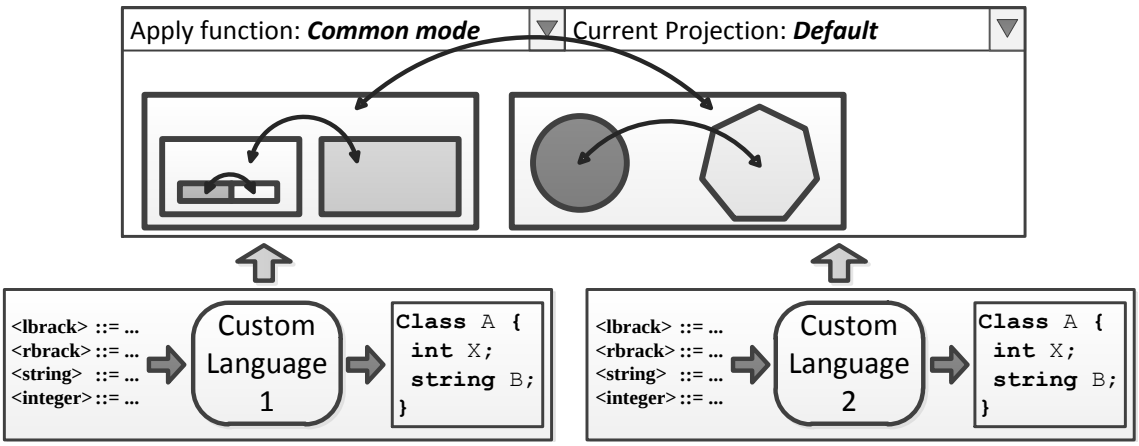


Figure 4.7: An editor can be extended with new languages that provide their description and projections in a suitable form.

The most challenging task is to integrate the proposed architecture of a projectional editor into Microsoft Visual Studio IDE. We have to bring up all the possible technologies that allow us to integrate new components into Visual Studio. Using one of them, we have to distribute an editor as a Visual Studio extension. The choice of a technology must be done in terms of an editor extensibility as well. The ability of extending it with new languages and projections must not be lost. Most likely this requirement will cause us to distribute the editor extensions in a similar form as the editor itself. These extensions will be integrated into the Visual Studio on a common basis, but they will be recognized and used by the projectional editor only.

≫ ***D8: Appropriate way to integrate the editor with its extensions into Visual Studio must be proposed*** *(see analysis Section 4.9, implementation Section 5.10)*

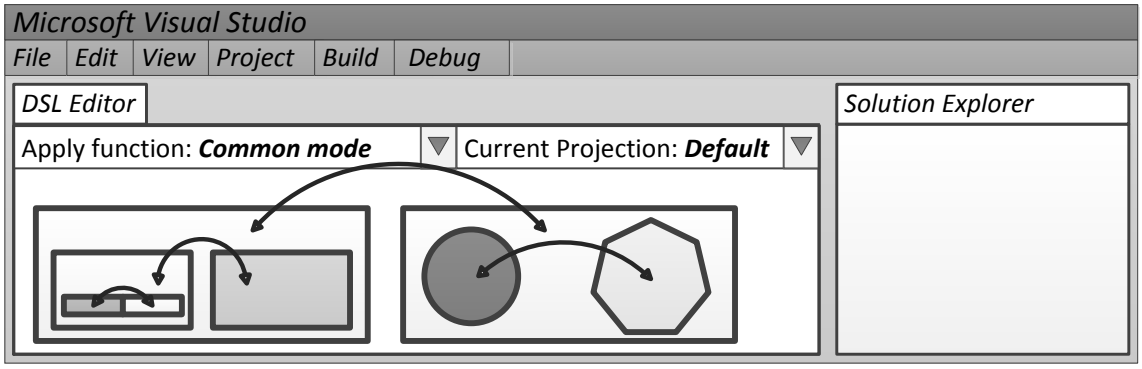


Figure 4.8: Proposed user interface integration - wireframe

Once all the fundamental decision steps are solved, we will face the problems with designing the projections themselves. Not all of them are trivial. In addition to the simple projections, we have to analyze how to design complex projections and what problems we have to deal with. We have to improve them of validations that indicate a syntactic problems have occurred and we have to find a way how to notify the projectional editor about such a problems.

≫ ***D9: The analysis of projections themselves must be done*** *(see analysis Section 4.10)*

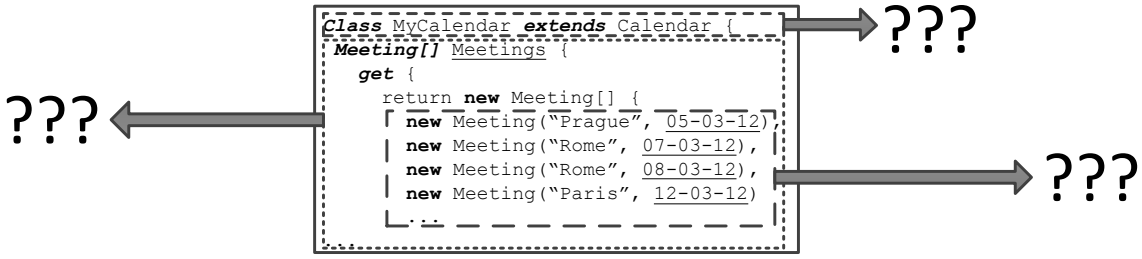


Figure 4.9: A projection built from partial projections of various complexity and a need to analyze them.

4.2 Language Description

This section refers to the decision step D1 in *Section 4.1*, the issue analyzed here is linked with an implementation in the *Section 5.3*.

A domain-specific language can be described in many ways. It is usually done by a kind of a formal grammar, via the Backus Normal Form (BNF) or using a meta-model. Since the form of a description will directly influence the further analysis, the choice must be done with a sufficient presence. A selected method must be clear and understandable in C#, either way we choose.

4.2.1 Meta-modeling

Source code of a language manipulates with several language constructs. A structure created using them is so called *model* of the language and contains basic language objects and their relations (for instance a subtyping). It is called *Abstract Syntax* since it does not contain a full definition of a language, its syntax, evaluation rules and typing rules. Nevertheless, the formal definitions reside on it. The structure of abstract syntax of code can be expressed by a tree representation, so called *Abstract Syntax Tree* (AST). Each node in this tree hierarchy refers to a certain language construct, let us call the root one *root language construct*. See an example of AST

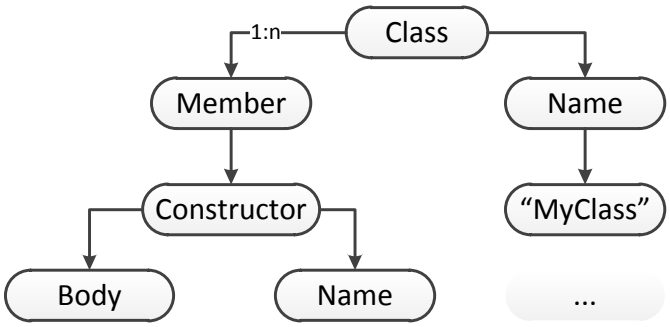


Figure 4.10: Example of an abstract syntax tree

on the *Figure 4.10*, Class is the root language construct of the hierarchy presented on this snapshot.

Source code can be considered as an instance of a model. In programming, a class defines a structure of an object and an object is an instance of a class. Hence, if a source code was an object, it would be an instance of a class that represents a model. Moreover, the structure of an object would correspond with an AST of the source code.

Meta-modeling language is a language that can be used to describe a language model. Various meta-modeling languages exist. However, if our language model was represented by a class, we should pay attention to the meta-models out of which the model can be generated in a form of class.

XML Schema (XSD) is a meta-modeling language that defines a structure of XML files. With the use of special tools, we can generate a class that represents the structure defined in XSD. Under given conditions, we can even backwards derivate XSD from the class. Thus the XSD and class may be mutually transferable.

It is obvious from the foregoing that if a language model was represented by a class, which may be transferable to the XSD, we could provide a language definition in a form of XSD. The source codes in that languages would be represented by instances of this class as well as the corresponding XML files. A root language constructs would be equal to the root elements of those XML files. Such a design will work if and only if the class representing the model will be serializable. Serialization is an ability of an object to be transformed into a XML stream conforming to a specific XSD. The class generated out of the XSD will certainly be serializable. The problem would have occurred if the language is defined by the non-serializable model only. But since we tend to define the languages by XSD primary, the serializabled model of a language will be generated in any case. The figure *Figure 4.11* demonstrates the design we have just discussed. XSD defines a language, which is described by a model in a form of C# class. A program of a given domain-specific language is stored in a XML file and is handled as an object in the editor.

Let us summarize the thoughts proposed in this section. One of the possible

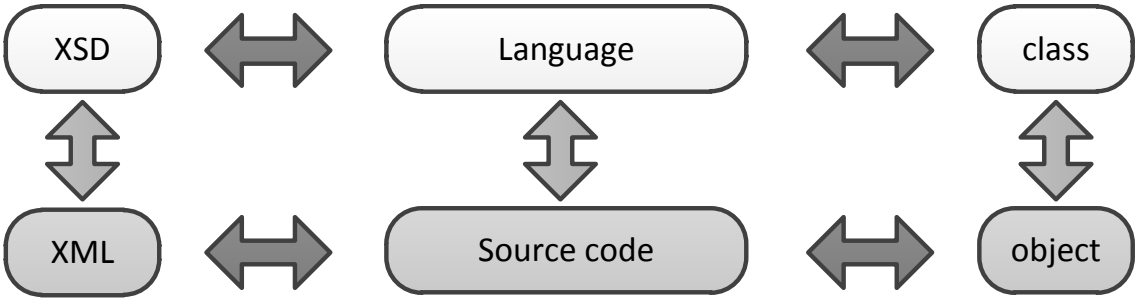


Figure 4.11: Various relations between language description and its representation

ways how to describe a language is a XSD meta-model. This method will bring us the following characteristics.

1. Language definition will be kept in a form of XSD
2. Source code of the language will be stored in a XML file
3. Source code is a XML and it can be therefore easily validated against its XSD language definition
4. The source code will be treated as any other C# object
5. The language definition does not cover the syntax of a language
6. Any C# object may be treated as a source code, even if it is not a source code at all. Just in case the object is not serializable, it cannot be stored to a file.

It is clear that the main problem is the lack of a syntax knowledge in the language model. It would have to be offset by adding this information to the model additionally. For instance, by extending the model of a method that would provide validation of each language construct against its syntax. The method would probably have to be implemented explicitly.

4.2.2 BNF Grammar Versus XSD Meta-modeling

Backus Normal Form (BNF) is a notation technique for context-free grammars. It is aimed at describing the language syntax and thus it covers a full language definition, which is certainly an advantage over the XSD meta-model. There are even powerful tools for building languages and generators for their reading, processing and executing. On the other hand, they do not give us the possibility to treat almost any object in the same way as a language instance. Such a possibility is crucial in the design of projectional editors, because it allows us to create projections of almost any objects and handle them as they were projections of a language. This benefit will be demonstrated later.

Despite all the shortcomings of the XSD meta-modeling, we decided to use it as a technique for the description of a language. The further analyzed design will be therefore tailored to meet its needs.

4.3 Projections Definition

This section refers to the decision step D2 in *Section 4.1*, the issue analyzed here is linked with an implementation in the *Section 5.4*.

Projection is a form of a visual presentation of a specific language construct. Language construct is represented by a graphical object drawn in the GUI. Therefore, such a graphical object is in fact a projection. Its complexity directly depends on a structure of the corresponding language construct.

We have to find a suitable way to define the projections and associate them with the language construct that are represented by them. In the *Section 4.2*, we stated the language is determined by its model whose instances are the sources written in it. We proved the model is represented by a common class. The problem of language projections is therefore reduced to the problem of how to define the projections of a class. It can be basically handled via an interface method, which the class is forced to implement. Such a method simply returns the projections of a given class where its members are presented in a convenient way. Nevertheless, this is more or less a technical solution that will be discussed in the *Chapter 5*. Now we are particularly interested in technologies that we can use to implement a specific projection, thus in general the technologies that produce a graphical output.

In .NET we have two options to deal with it, either using aged Windows Forms (WinForms) or XML based Windows Presentation Foundation (WPF). If we need a direct access to the native Windows API, the WinForms is probably the best choice for us since it supports it natively. There are definitely a few advantages of WinForms over WPF. It is easier and a bit more stable, because of its age. The application would run even if the only .NET framework installed on a machine is of version 2.0. For instance, Windows 2000 is not supported by a newer version of .NET at all. However, If we are running a fresh version of .NET framework, the benefits of MVVM based WPF will probably outweigh the advantages of WinForms and thus become crucial for us to decide.

4.3.1 Data Binding, an Essential Feature

A projection is nothing but a way of arranging the basic graphic elements on the screen. The only thing missing is a value presented in that projection, a value of a property of a class instance we visualize. This value must be therefore injected to the projection and must be kept up to date with regard to its changes within the frame of projection. On the other hand, the projection itself must be also kept up to date with respect to current property value. Any changes in the value of the property must be automatically reflected in the projection, which is forced to redraw. This may not seem like a major problem, however, there are usually more

projections of a class property and all of them must be synchronized with the value presented. This is in fact the very essence of projectional editors.

Data binding is a mechanism that binds two data sources together and keeps them synchronized. In our case, one source is a class property, the second one (target) is a property of a projection. Such a mechanism seems to be indispensable for those designing projectional editors.

4.3.2 MVVM Pattern

Data binding is essentially an association of one language construct with another in a projection. A language is represented by a model and a projection is a graphical object placed on the view level of an editor. Hence the data binding is a relation between a model element and a view element, thus it tends be the MVVM design pattern.

Such a pattern has a number of benefits. The clear separation of a model (business logic) and a view allows us to create unit tests of the model independently of the view. Therefore, we can provide the model of a language with the unit tests regardless the projections defined for it. Moreover, an instance of a language is a serializable object representing a source code. Therefore, when the editing of a source is over and we are about to save it, the object is serialized and stored to a XML file. If the model was not separated from the view, the saving process would probably be trying to save the information of projections as well. That would mean that the metadata representing a visual appearance of a language would be attached to the sources itself. Since we prefer to keep the source files clean, untainted with the information of their visual appearance, we tend to use MVVM in the design of projectional editors.

4.3.3 WPF Wins

WPF is a toolkit that has been primarily aimed in the use of MVVM pattern in the application design. It naturally supports data binding of the model elements to the view elements. The view part is described by a XML based language called XAML. Although the direct access to the Windows API is not supported, it can be achieved via hosting the WinForms elements in WPF. Since the access to the Windows API will not probably be the common request, the sufficient information should be that it can be reached using WPF as well. Moreover, WPF provides a higher level of a freedom during the graphic customization. It can be worthwhile, since almost any control look and behavior can be rewritten using a XAML template, which is in fact a kind of a projection. WPF thus becomes the main technology during the design of our projectional editor.

4.4 Projective Object

This section refers to the decision step D3 in *Section 4.1*. The corresponding implementation can be found in the *Section 5.5*.

As we outlined before, the projections are bound to the model of a language by a technique called data binding and any complex data type or a class can be seen as a model. The question is in what form the projections can be attached to the model.

In general, there are two possible ways to handle it. We can either design an independent structure to hold the projections of a model, or we can keep this knowledge directly in the model and thus extend the model appropriately. Both the approaches have pros and cons, these will be discussed below.

A Separation of Model and Projections

When we separate a model from its projections, the model stays clean, shield from everything what has nothing to do with the model itself. It just does not care that it might be used in a specific context, like in the context of a projectional editor. On the other hand, any time we need to do a projection of its instance, we have to find out if there is any projection defined and where do we get it. Thus we additionally need to keep a mapping between the model and structures describing its projections. Furthermore, we have to propose a mechanism to maintain such a mapping. The *Figure 4.12* demonstrates this approach.

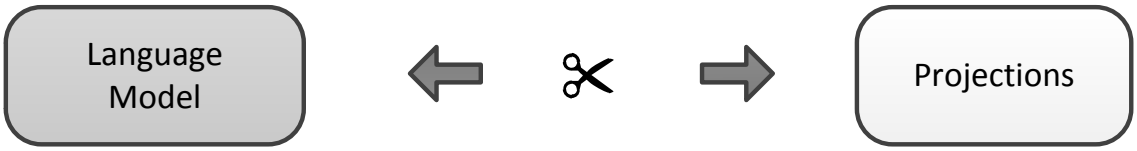


Figure 4.12: Model and projections are separated

Merging of Model with Projections

The approach of merging model with projections would make the model *self-projective*. Each of its instances would keep the information about its own visual forms. We can achieve that by making classes participated in a model to implement a method which gathers the desired projections. Therefore, it is reasonable to design an interface, which adequately conveys the projections of a class. This interface must be implemented by all the classes requiring to display themselves in a projection. All defined projections will be aggregated to the collection and returned by that interface method. Let us call the interface *projective interface* and a class that implements it a *projective class* or a *projective type*. An instance of this class is called

projective object. See the demonstration of this approach on the *Figure 4.13*.

Figure 4.13: Projections being kept inside a language model

Needless to say that a usual class contains properties of various data types and so does the model. Any property whose data type is a class or a collection of class types might be considered as a sub-model and thus presented in a projection. If these classes were projective classes, their instance would be again projective objects. Hence the values of properties within a class would be projective objects. Any class participated in the model can be promoted to the projective class and thus becomes able to be presented in a projection. This leads us to a hierarchic structure where a projective object may be a part of another projective object and so on. Therefore, within a projection may be sub-projections of elements contained in the corresponding model (see the *Figure 4.14*).

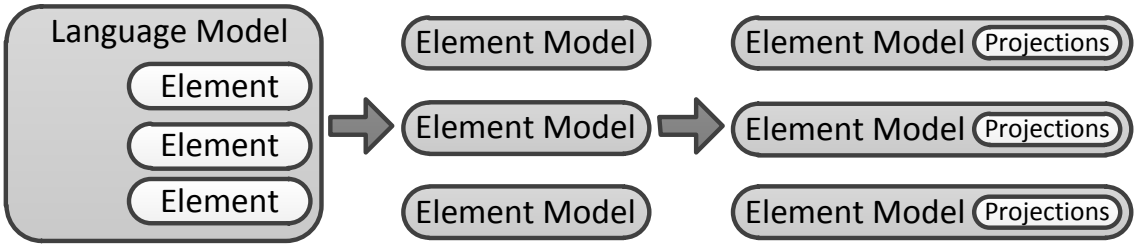


Figure 4.14: Language model composed of various elements (language constructs). The complex ones can be seen as models themselves and enhanced of projections.

A hierarchical organization of projective objects brings us another problems we have to be aware of. If a same element is presented in more different projections, we have to establish a mechanism to refresh it in all projections whenever it is changed in any. Otherwise, it might happen that in one projection an out-of-date value of an element is displayed while another projection already updated it.

Merging Wins

Since there seems to be more classes which have to be enhanced of projections, the second analyzed approach seems a bit more suitable for us. Despite the fact we will mess up a model with seemingly unrelated content, we will not be struggling with a mapping of an additional structure to our model and its maintenance. Any time we need to have projections of a certain class, we just simply implement the given interface. Moreover, there is a technical trick that removes the shortage of messing

up the native model with projections. We will deal with it in the *Chapter 5*. The idea of projective classes is on *Listing 4.1*.

The hierarchy of projective objects mentioned above requires the existence of a component to manage their projections. This is so called projective component and it is going to be discussed in the next Section.

```
public interface IProjective {
  List<Projection> GetProjections();
}

public partial class Class : IProjective {
  ...
  public IEnumerable<Member> Members { get; set; }
  ...
}

public partial class Method : Member, IProjective {
  ...
  public List<Projection> GetProjections() {
    ...
  }
}

public partial class Attribute : Member, IProjective {
  ...
  public List<Projection> GetProjections() {
    ...
  }
}
```

Listing 4.1: Projective classes

We analyzed that the injection of projections into a model is likely to be useful. We outlined that these projections can be accessible via the projective interface, which may return them in a collection. Nevertheless, there is a goal G4a which requires the hops between projections in dependence on events that occur in a projection. It means that a projective object must be aware of relations between events and projections. A single projection is completely isolated from another projections. It does not even know that another projections exist. Thus a relation between events cannot be kept on the level of projections. Higher level object must be probably returned in a collection, instead of a pure projection. For this reason, the *Listing 4.1* is presented just because of an idea and concept. In the next section *Section 4.5*, we will modify the projective interface a bit.

4.5 Event-driven Projections

This section refers to the decision step D4 in *Section 4.1*. The corresponding implementation can be found in the *Section 5.6*.

Event-driven projections arise from a goal G4a stated in the *Chapter 3*. If a

routed event occurs (see details about routed events in [16]) in a projection, another defined projection should appear instead. For instance, let us say, there are two different projections, a *BeforeClick* projection and an *AfterClick* projection. Whenever a user raises a *Click* event within the *BeforeClick* projection, the *AfterClick* projection should appear instead of the original one. Such a relation *[Click, BeforeClick, AfterClick]* is so called event-driven projection.

It is obvious that the concept of event-driven projections may result in a kind of a workflow among certain projections. Since the projections are not aware of each other, this workflow must be described over them. The fundamental question is, whether it is reasonable to allow hops between any two projections or just inside a group of projections.

Hops Between Any Projections (The Anarchist Hops)

It might be a tempting feature to allow hops between any two projections. On the other hand, sometimes it is reasonable to have a set of projections that hop between themselves and no other projection should have a possibility to interact with them.

For instance, let us say there are two projections, a graphical and a textual one. When we click the graphical projection, a graphical element would animates, when we click the textual projection a text would be highlighted. The after-click moves from a textual projection into a graphical one does not have much sense, because we cannot probably animate the text with the same graphical animation. Moreover, we want to choose initially the original graphical or a textual projection only, not the ones, which appear after we click the initial projections.

Thus we would need to define, which projections are available as initial projections and which projections may appear only after a certain event is raised. Additionally, we would need to prevent hops between specific projections and therefore provide a mechanism, by which we could describe projections that can hop between each other and that cannot do so.

Hops Inside a Group of Projections (Intra-group Hops)

Under this concept, projections are split in isolated groups and each group has its own workflow. Inside the group, the projections can hop between each other however they like. A user is provided with the groups of projections and he can choose a group which he wants to interact with. A certain projection inside a group is considered as an initial one and in fact represents a group. For instance a *textual group* contains one initial projection via which we can enter a workflow of another textual projections inside the same group.

Intra-group Hops Wins

Since the design of anarchist hops seems to be unclear and slightly complicated, it tends to a concept of intra-group hops of projections, which live their own lives and are driven by an internal workflow. Under this concept the groups of projections must be described appropriately as well as the workflow inside a group. Thus we put projections into so called *projection containers*, which are described below.

4.5.1 Projection Container

Projection container is an independent unit of projections, which is extended of a workflow among them. Workflow is driven by events that may occur inside a specific projection. The way how the workflow is described directly depends on the implementation and it is therefore discussed in the *Section 5.6*.

In this section, we will discuss, how the concept of projection containers affects the projective interface, which initial version was proposed in the *Section 4.4*.

At first, we expected that a list of available projections will be attached to a certain language construct. In this chapter, we analyzed that each construct should be better provided with independent groups of projections, whereas exactly one projection in a group is considered as initial and exactly one projection is active. The *active* projection is the one which should be displayed with respect to the workflow. In that case a projective interface must change correspondingly as on the *Listing 4.2*.

```
public interface IProjective {
  List<IProjectionContainer> GetProjectionContainers();
}
```

Listing 4.2: Projective interface

The idea of model enhancements would not change at all. Each class participated in a model may be promoted to a projective object just by implementing the projective interface. Since this section is just an analysis from which the concept of projection containers has emerged, we will not continue to pursue the implementation and we will leave it to the *Section 5.6*. See the *Figure 4.15*, which represents the concept of projection containers associated with a language construct.

4.6 Projective Component

This section refers to a decision step D5 in *Section 4.1*, the issue analyzed here is linked with an implementation in the *Section 5.7*.

When we stated a decision step D5, we thought that a language construct will

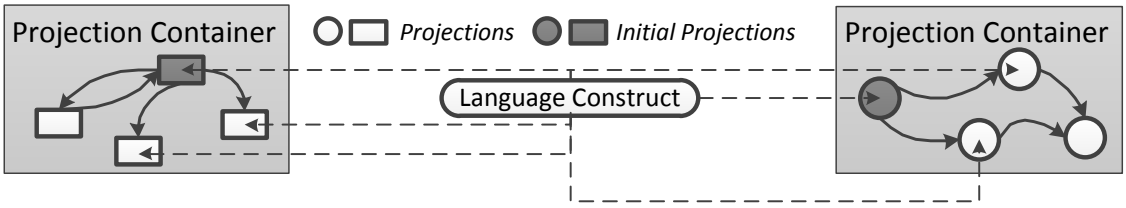


Figure 4.15: Language construct provided with several projections organized into two containers

be provided with several projections and a projective component bundled with a language construct will be a component responsible for swapping them. However, in the *Section 4.5*, we concluded that it is likely to be useful to group projections into so called projection containers, where exactly one projection of a group is picked with respect to the workflow inside container and displayed afterwards. Such a group of projections represents projections that are kind of similar. It simply makes a sense to switch from one projection to another. Since several groups of projections can exist for one language construct, a job of a projection component is to swap between the groups, not between the projections itself.

Projective component can be then considered as a graphical control whose the only and a main job is to enable the user to choose between available containers and select one to display. Since in the WPF terminology, the graphical objects are so called Controls, we will use a term *Projective Control* in the same context as the Projective Component. Therefore, these terms will coincide in the further analysis.

One of the overall goals of this thesis is to implement a projectional editor. The projectional editor maintains the code of a language composed of various language constructs. Each language construct is provided with few groups of projections and these can be handled via a projective control. Hence it seems that a projectional editor is a control that is composed of several projective controls. However, the projective control is a universal component. It does not necessary need to be wrapped by the editor only. It can be used in a context of any WPF application that requires to switch visual controls within the GUI[1] while interacting with a user. Thus it should be designed on a general level to host any WPF content. The projective control as a universal component is demonstrated on the *Figure 4.16. Projectional Editor Component* drawn on a picture might be confusing a bit, that will be clarified in the *Section 4.7*.

4.6.1 Structure of a Projective Control

Projective Control is initially just an empty container, which requires to be fed up by projections of a specific language construct. These are organized to projection

[1]Graphical User Interface

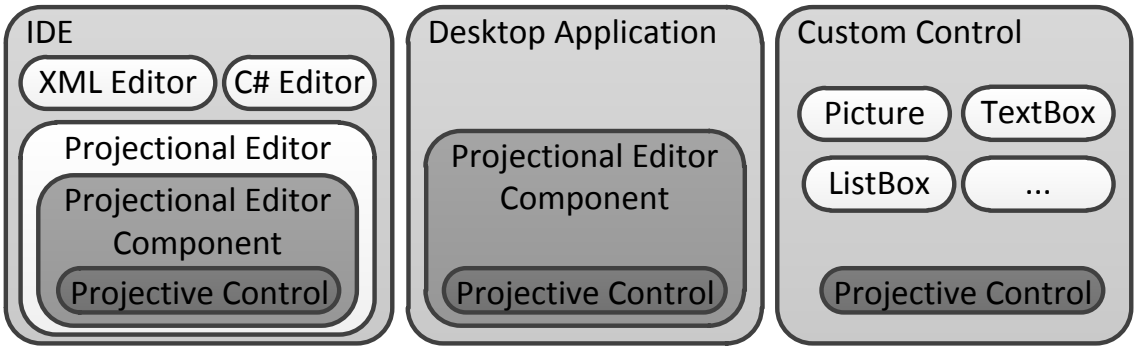


Figure 4.16: Projective control used as a universal component in a context of IDE, desktop application and custom control

containers. All these containers are then supported by a projective control. It means the user is able to switch among all the containers being defined and can directly edit a part of the instance of a language model within the active projection of a container.

A runtime instance of a model is in fact a source code of a program that is implemented using a designed language. In order to let the projective control visualize a part of the source code in a specific projection (diagram, highlighted source code, etc.), the corresponding projection must be created before. Additionally, all the elements of a projection intended to present any part of a model need to be bound into it. These projections are grouped and returned in a collection of projective containers via the projective interface. It would be useful to put this collection directly to the input of a projective control, but this would undermine its universality. A universal component knows nothing about a collection of some projective containers since it is simply universal. For this reason, the component must be wrapped by a less universal one. The one tightly coupled with projections and aware of projection containers. Its task is to adopt projection containers for the use in a general projective component. We will call this wrapper *Projective Object Control* and we will discuss it in the next section.

4.6.2 Projective Object Control

Projective Control is a general purpose component. It knows absolutely nothing about the projective object and its ability of self-projections. An additional control called *projective object control* is designed to adapt projective object to the projective control. It grabs the projections from projective object, adapts them and puts them on the input of a projective control. The process of adaptation is purely technical and it will be discussed in the *Subsection 5.7.2*. See the *Figure 4.17* for a demonstration.

Figure 4.17: Projective object bound into projective object control requires a proper adaptation of its WPF projections

4.7 Projectional Editor

This section refers to a decision step D6 in *Section 4.1*, the issue analyzed here is linked with an implementation in the *Section 5.8*.

Projectional editor is an editor of a given domain-specific language source code. It must be aware of the language structure and its projections to maintain the given source code appropriately. We stated that it consists of a projective components and its only job is to maintain them in global with the use of various features. Among these global features, there are definitely two important ones, loading and storing the file. No matter where the file comes from, its content must be cut up into pieces and these must be loaded into appropriate projective components. Similarly the content of a file must be constructed before the file is saved.

Furthermore, an editor can be either a standalone application itself, part of another application or it can be integrated in a given IDE. It is evident that it must be treated as a universal component usable in various contexts.

A deep analysis of such a component is an essence of the sections below.

4.7.1 Hierarchy of Projective Components

We said that an editor provides features that can be applied globally on all underlying projective object controls. The fundamental question to answer is, how it may happen that there are underlying projective object controls. We will clarify that below in this section.

Under the projective object control that corresponds with a certain language construct, there is in fact a hierarchy of projective object controls. The projections of that construct are composed of various graphical controls. Thus they can contain the projective object control since it is a graphical control as well. This underlying control refers to a certain language construct that lies lower in the AST hierarchy and visualizes it in one of its projections. This mechanism results in the hierarchy of projective controls in accordance to the AST.

The *Figure 4.18* outlines the projective object controls that are organized in

accordance to the AST. Rectangles represent projective object controls of a corresponding language construct in the AST hierarchy. For instance, a projective object control A' is created for a language construct A. A provides several projections that visualize its direct successors B and C. Both of them are language constructs and are provided with a set of projections, thus a projective object controls B' and C' are created and placed inside the parent projection. This controls are underlying controls of the projective object control A'.

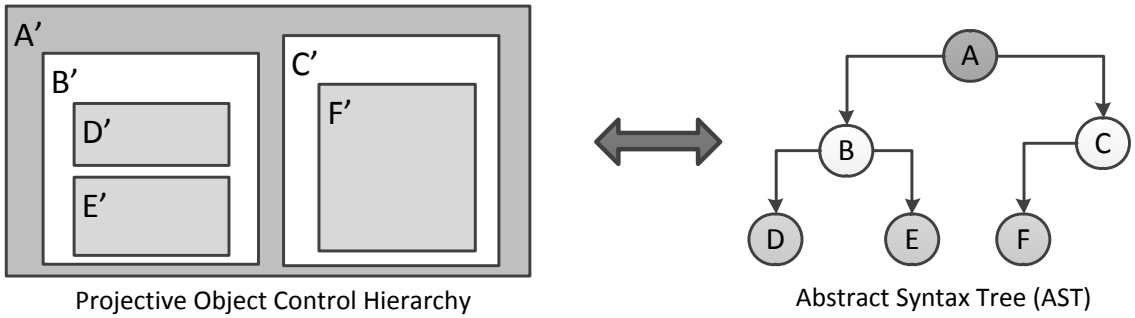


Figure 4.18: Hierarchy of projective object controls. Each node in the AST tree corresponds with a projective object control on a certain level of hierarchy.

The purpose of this section was to demonstrate that projective object controls are organized in a hierarchy. At the same time, more of them can be displayed in an editor, thus it could make a sense to maintain them globally in a convenient way.

4.7.2 Projective Control of a Root Language Construct

Since we are about to implement projectional editor as an independent component, we have to consider how it should look like. We will show that under certain circumstances, it can be created with the use of one projective object control only.

A root language construct is represented by a class participated in the model. If this class implements a projective interface, its instance is a projective object, let us call it *root projective object*. Its projections are a kind of the main projections of a language. They manipulate with the whole source code at once. We discussed that a projective object is handled by a projective object control, which maintains its projections. If the root projective object was bound to a projective object control (*root projective object control*) which is wrapped by another component, such a component enriched by certain features would become a projectional editor, let us call it *projectional editor component*. Its features would serve to maintain the global state of all the underlying projective object controls.

See the *Figure 4.19* which illustrates the binding of a root projective object to a projective object control and its wrapping by a projectional editor component.

It means that if a root language construct is projective, its projective object control will contain any other projective object control. It can be wrapped by a

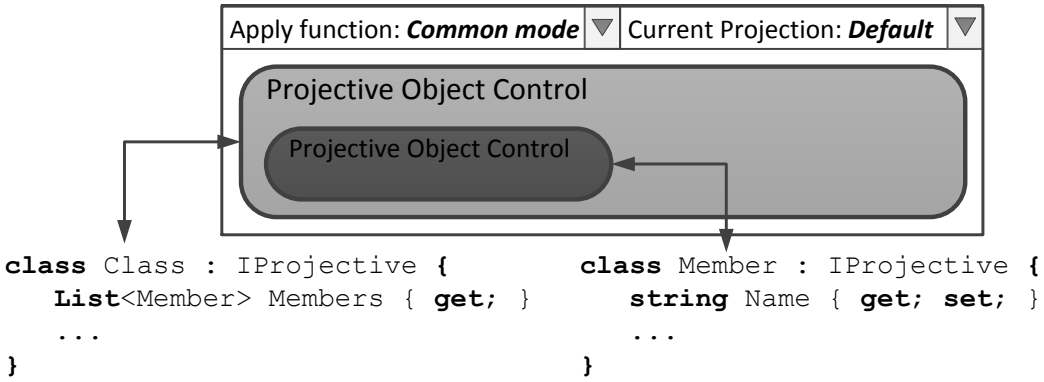


Figure 4.19: The hierarchy of projective object controls belonging to the classes in a model is wrapped by a projectional editor component

new control, on whose level, we can handle the global functions applicable to all projective object controls at once.

4.7.3 Projectional Editor Component

An IDE editor is more or less a graphical control wrapped by some specialized class that communicates with the rest of IDE through a certain IDE interface, which it implements. That graphical control is a component with its own core logic and GUI part for the interaction with a user. It is kind of isolated within the IDE. Any request for the communication with another IDE parts must be handled by its wrapper.

One of the overall goals we have is to implement a projectional editor within a selected IDE. Such an editor is composed of projective controls and thus it is a graphical control as well. We called that graphical control Projectional Editor Component. Since we want to keep it universal, the IDE-specific tasks must be handled on the level of its wrapper and appropriately communicated to the underlying projectional editor component.

For instance, an IDE part aimed at opening, saving and closing files can notify the editor wrapper through the IDE interface that a specific file of a given name is opening. A wrapper will communicate the name of required file to the projectional editor component, which is responsible for loading the file, deserializing[2] it into a projective object and presenting it using a projective controls.

In the opposite direction, if a projective object is changed within the projectional editor component, this information must be communicated to the wrapper. The wrapper must notify IDE via the IDE interface that a file has changed and should be marked as dirty.

[2]Serialization is a process of translation a specific data structure into a format storable to a file. Deserialization is an opposite operation.

It seems that a universal projectional editor component must be provided with a suitable interface, which can be used for the communication with any of its wrappers. So far we have discussed various components, which we operate with. The *Figure 4.20* should make a clear overview, how these fundamental components are leveled.

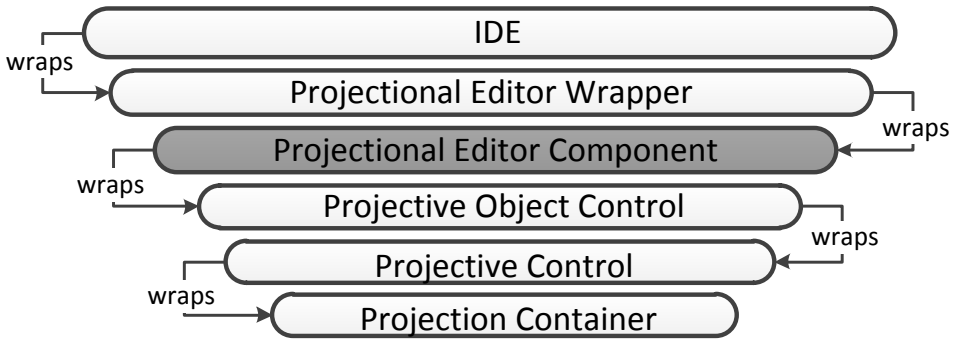


Figure 4.20: Components of the proposed design organized in successive levels. Projectional editor component is significantly marked in the hierarchy.

In this section, we evaluated that a projectional editor is represented by an independent projectional editor component. It provides a well-defined interface to handle the operations coming from outside and apply them to all projective components in global. An interface is more or less a technical issue and we will have a look on its implementation in the *Section 5.8*.

4.8 Editor extensibility

This section refers to a decision step D7 in *Section 4.1*, the issue analyzed here is linked with an implementation in the *Section 5.9*.

Extensibility is an ability of the editor to work with an information that is not a part of the editor itself. It must be injected to the editor and thus extends it of such a knowledge.

Since we cannot cover all existing languages and corresponding projections within one projectional editor, we need to find a way to extend an editor with a new custom language. First, we must evaluate what information we are actually missing in the editor. In the last *Section 4.7*, we introduced a projectional editor component and we mentioned that it must provide a well-defined interface to be shielded from the complexity of an environment where it operates. Via this interface, a request for file opening must go through. Projectional editor component will deserialize the file to the projective object and grabs its projections. The fundamental question is, how the editor knows the data type of an object to which the file can be deserialized. The data type of a projective object is a projective type. Thus the projective type

related to the given file is the missing information required by an editor (see the illustration on the *Figure 4.21*).

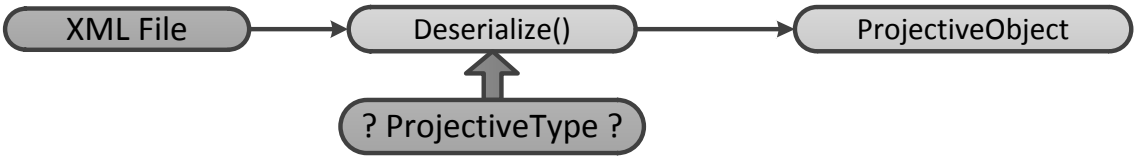


Figure 4.21: Projective type missing for a deserialization of the input XML file to the corresponding projective object

In addition to that, an editor must be able to recognize, which projective type to use for deserializing a certain file. For this reason a XML file must provide a key that uniquely identifies the language of a file (for instance a file extension could be a good candidate for such a key). The key coupled with a specific projective type will create a relation between the file and a projective type suitable for its deserialization (see the *Figure 4.22*).

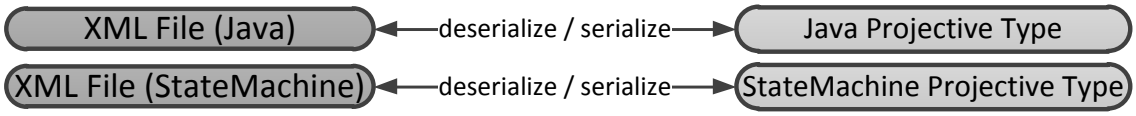


Figure 4.22: Relation between a file and a suitable projective type, which the file can be deserialized to

From the analysis above, it seems that if a projectional editor component is dealing with the opening of a file, it must find the relation between the file and a projective type. Afterwards, it can deserialize the file to a proper projective object (root projective object) and present it in a root projective object control as described in the *Subsection 4.7.2*. Thus the information of all projective types and their relations to the XML files must be kept in a convenient form and injected into a projectional editor component. We will call this form an *add-in* and we will take a look on its implementation in the *Section 5.9*.

4.9 Visual Studio integration

This section refers to a decision step D8 in *Section 4.1*. A corresponding implementation can be found in the *Section 5.10*.

Our overall goal is to provide a projectional editor in a form of common Visual Studio editor. To achieve it, we have to evaluate what such an integration involves.

First we need to determine the form in which an editor will be integrated to MSVS. This decision will strongly influence the entire integration. We will deal with that in the *Subsection 4.9.1*.

In the *Section 4.8*, we concluded a need of creation so called add-ins that will support the editor with a knowledge of a new language. We have to distribute this knowledge appropriately in an acceptable form for MSVS. We will discuss the add-in distribution in the *Subsection 4.9.2*.

Visual Studio is a development environment composed of several designers and tools that interact with each other. Thus our editor cannot be just an isolated component within the IDE, it must communicate with other well-known parts of MSVS (for instance error viewer). We must distinguish components with which is suitable to cooperate and find a way how to establish such the cooperation. We will scrutinize it in *Subsection 4.9.3*.

4.9.1 One-language-only Versus Multiple-language Editor

The contribution into Visual Studio environment can be done in various ways. We can extend the official WPF-based code editor with new features that support projections, or we can implement our own editor and deploy it to MSVS using VSIX package[3].

Since the common code editor of Visual Studio is written in WPF and it is slightly configurable, we can enrich it of new features quite easy. On the other hand, we do not have a full control of such a customization which could be quite a limiting restriction.

Thus we end up with an implementation of VSIX package containing a component that represents our editor. In our case, things are a little more complicated than we anticipated. An editor must be extendable with new domain-specific languages. These languages are custom, created by a developer who insists on maintaining their sources in certain projections. Therefore, we can either release many language-dependent projectional editors as a VSIX components, or we can provide rich empty projectional editor that cooperates with another, later created VSIX components that distribute the add-ins. Both the approaches seem to have the same result, thus we have to evaluate them deeply.

4.9.1.1 One-language-only Editor

One-language-only editor is actually not an editor extendable with additional domain-specific languages. It is just a workaround to achieve the same goal - implement a MSVS projectional editor for the given DSL of a custom projections.

Under this design, an editor hosts exactly one DSL, we do not have to worry about a projective type to be used for input file deserialization. The type is simply uniquely determined. Each DSL defines the extension of its source code file, which

[3]VSIX package is a deployable unit containing various MSVS extensions

the editor is paired with and an IDE opens the editor for any file of this extension.

After closer inspection, we find that there is no need to inject the projective type in any sophisticated way, since it is well-known type and can be easily instantiated in the projectional editor. Thus the way of file deserialization is uniquely determined.

This is definitely an easier solution to host custom DSL editors in the IDE, but it brings a number of drawbacks like multiple instances of the same projectional editor over the whole environment that differ only in the add-in with which they cooperate.

This might significantly affect the maintenance of the projectional editor, since any new version of the core projectional editor component would require to be updated in all already deployed editors. This is for sure much less flexible than an update of only one editor, a thereby affecting all supported DSL.

Furthermore, the deployment of a whole editor into the MSVS is a bit more clumsy than just an extension to the existing one. Moreover, when we consider the increase in the functionality and features of the projectional editor itself, the existence of several one-language-only editor will be just a nuisance.

These drawbacks are serious enough to reject the one-language-only design. Its illustration is on *Figure 4.23*.

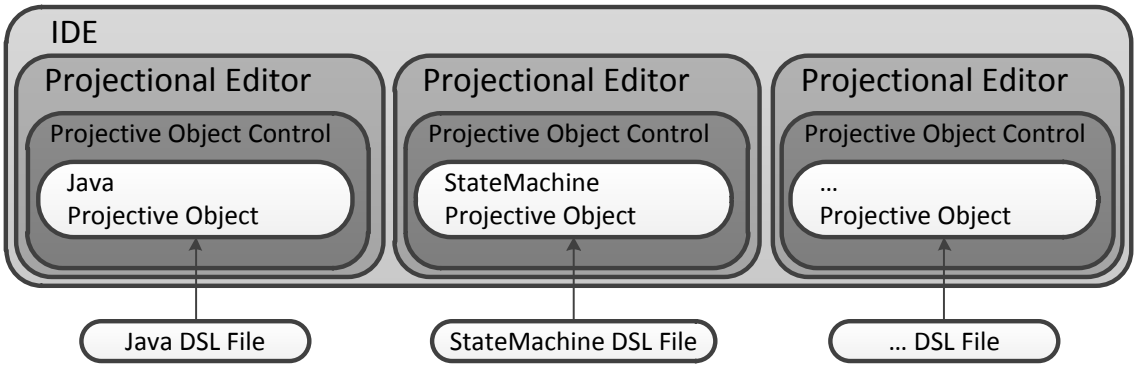


Figure 4.23: One-language-only editor IDE integration: Each DSL is provided with its own editor intended for the opening files of the given DSL.

4.9.1.2 Multiple-language Editor

Multiple-language editor is an opposite approach to implement extendable editor. It resides on an existence of a projectional editor that is supported by custom domain-specific languages exported via MEF from another Visual Studio parts. Such an approach is facilitated by a native support of MEF within the MSVS 2010. Its design and benefits are discussed below.

The multiple-language editor eliminates the deficiencies of the one-language-

only editor. It is maintainable and lightweight since it contains only general functionality. The DSL-specific functionality, as its model and projections, is injected by a MEF component. Therefore, there is the only one projectional editor installed in the IDE and each DSL, with its projections is deployed in a form of an extension into that projectional editor. Under this design, the editor can be a rich application in terms of features and implemented functions. Furthermore, any bug-fix or upgrade is just a matter of an update of one MSVS component, which affects all the supported DSL. On the other hand, if there is a custom request of one DSL, which needs to be treated on the editor level, the general functionality must be added. So the lightweight design might disappear.

In spite of this drawback, several valuable benefits are captured. The MEF component that extends the given projectional editor can be simply injected into MSVS without a need for additional changes. Moreover, an editor, which the MEF component is injected to, does not necessary need to be an IDE editor. The projectional editor, as a part of the standalone application, would operate completely the same way.

After the review in the previous two sections, we decided to implement a multiple-language editor in a frame of this thesis.

4.9.2 Add-ins Deployment

In the *Section 4.8* we ended up with the conclusion that we have to create so called add-ins to hold the projective types and their relations to specific XML files. Their design is more or less a technical issue, thus we postponed it to the implementation related *Section 5.9*. It is clear that the add-ins will be a kind of a C# class. Regardless their structure we have to distribute them in a form eligible to be deployed to MSVS on a common basis.

In this section, we will analyze these units of add-ins deployable to MSVS. Although the analysis done so far tends to use MEF, we will comment a concurrent technology MAF first.

4.9.2.1 MAF Add-ins

MAF has been touched in the *Subsubsection 2.4.3.2*. It is a higher-level framework that targets to the development of applications extensions. It is a complex tool for the treatment of application extensions. It provides features like *add-in discovery*, for the sophisticated seeking for available add-ins, *add-in activation*, for the controlled activation of an add-in, *isolation levels*, for the add-in handling within an application domain, *Lifetime Management*, for a thought management of an add-in lifetime, and several others. Thanks to all of these features, even basic establishment and

configuration of a solution dealing with MAF becomes a way harder than using MEF.

From the foregoing it is clear that MAF is probably a good tool for a creation sophisticated add-ins, but since we need basically a simple deployment of projective types and their subsequent injection to already installed MSVS component, MAF seems to be an overkill for us if most of its potential remains unused.

4.9.2.2 MEF Add-ins

MEF was introduced in the *Subsubsection 2.4.3.1*. It is a set of libraries, which enable the easy extension of application components in runtime. An application may therefore use the extensions even without hard-coded references or without a fragile configuration file. Only the extension metadata are usually loaded and thus the instantiating of an extension itself may be postponed, as well as the load of assembly, where the extension is defined. MEF is based on the composition of application components. Each component declaratively specifies its *Imports* - another components, which it depends on, and its *Exports* - capabilities, which it provides to another components. All the Imports are satisfied with the available Exports immediately after the MEF composition engine proceeds with the composition. The *Figure 4.24* demonstrates the composition of MEF components within an application, IDE, or across the applications spread in the system.

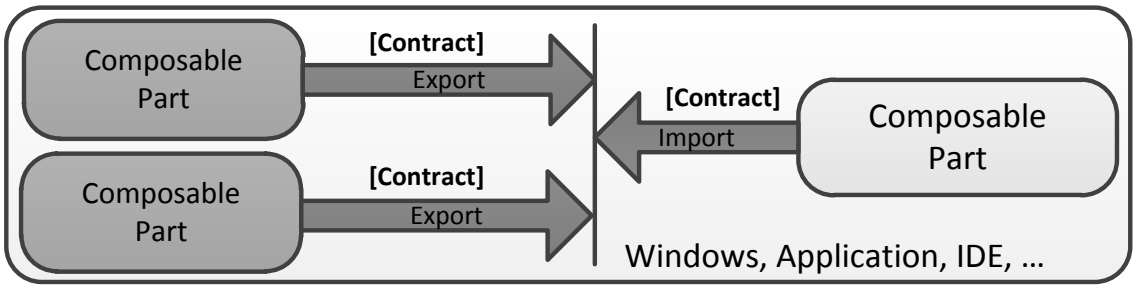


Figure 4.24: MEF composition across applications: Each composable part can be exported or imported from a different application, an overall composition is happening using required contracts.

For our purposes, MEF is a suitable framework. It can export any class implementing a specific interface and even provide us with its metadata. In metadata, we can distribute the projective type itself and thus postpone the instantiation of an add-in till the time we actually need it. Since a projective type is mainly the information we are interested in, we can even completely avoid the instantiation of an add-in.

In the opposite direction, a projectional editor is just importing MEF components implementing a specific interface and thus it is aware of all projective types exported by another MSVS components.

The only challenge that remains is how to wrap the add-ins by VSIX components and export them properly by MEF. Since this is a technical task, we will discuss it in the *Section 5.10*.

4.9.3 Association with MSVS Components

Microsoft Visual Studio is IDE. As well as other IDE, it facilitates the development using comprehensive tools. These usually cooperate with each other. They provide a general functionality that might be used by other tools. For instance a *Properties Viewer* is aimed at editing properties of a specific entity. A XAML designer uses it to edit properties of currently selected control and similarly a *Solution Explorer* maintains there the properties of a selected resource.

Since projectional editor is about to be a part of MSVS, it seems that it should not be completely isolated and it should cooperate with certain MSVS parts. Below, we will list components, which it is suitable to cooperate with and we will briefly comment them.

4.9.3.1 Error Viewer

Error List is in general a view of errors, warnings and info messages that has appeared in the compilation or precompilation time. Any problem that occurs in a projectional editor can be reported to the *Error List* and displayed properly.

It is apparent that a notification of messages is IDE-dependent, thus it cannot be invoked from the projectional editor component since it is a universal one. The notifications must be treated in the wrapper of projectional editor component, which must be aware of all problems to notify. In the *Section 5.10* we will take a look how such the notifications can be implemented.

4.9.3.2 Templates

To speed up the development of add-ins to our projectional editor, we have to provide an add-in project template. A project template will create MSVS project that contains fundamental resources and convenient directories structure.

Together with that, a template and a suitable wizard for the creation of new domain-specific language files must be implemented. Since new domain-specific languages are injected to MSVS via add-ins, a wizard must interact with them appropriately.

4.10 Anatomy of Projections

This section analyzes a decision step D9 stated in the *Section 4.1*.

As noted many times before, the projection is just a way how the source code of a given DSL is presented to the user. We decided to implement the projectional editor as a .NET component with the use of WPF, thus projections themselves will be created using this technology as well. The *Section 5.4* deals with the technical aspects of WPF like data binding, templates, etc. Thus the implementation details of projections can be found there. In this section we will discuss projections on a theoretical level. We will categorize them and analyze what behavior is desirable. Finally, we will evaluate, whether general projections applicable across different DSL languages can be designed.

For a simplicity, a projection is purely a graphical control which is bound into a model of given DSL with the use of a technique called data binding. Data binding maintains the synchronization of a visual control with a data it represents. Without any further explanation, we will understand binding as a connection between graphical appearance of a language and its data representation.

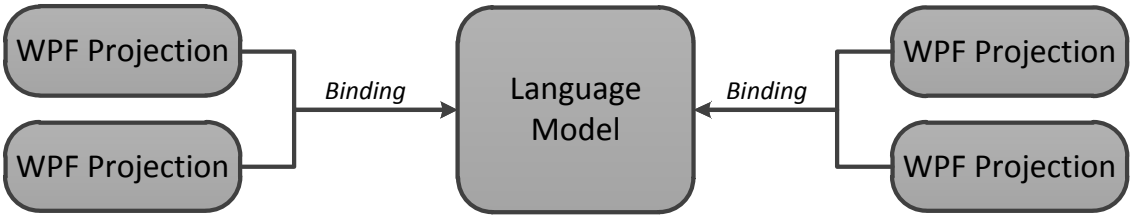


Figure 4.25: WPF-based projections bound to the language model

4.10.1 Projections of Elementary Data Types

The binding can in general tie up a data object of any type with any graphical object if it supports it. Elementary types are usually represented by a common graphical controls. Strings, numbers or dates are presented in a form of text boxes, numeric text boxes or date pickers. Collections of these types are often bound into list boxes, combo boxes or grids. The listed controls are basically projections of elementary data types. Since we need them over and over again, it makes a sense to provide a factory[4] to ease their creation. The projections of complex data type will arise from their composition. The *Figure 4.26* illustrates projections of elementary data types.

On the left, there is a model of a DSL, which is linked to the projection via bindings. For instance a WPF projection contains a date picker intended to edit a *Date of Birth* of a person. The date picker element is bound into a corresponding

[4]Design pattern for the instantiation of objects

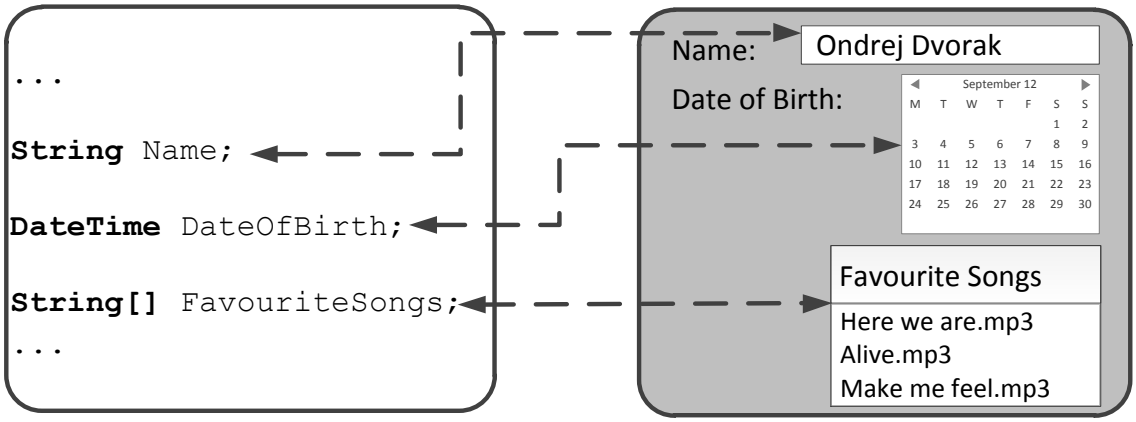


Figure 4.26: Elementary projections of Strings, DateTime and String array are represented by a text box, date picker and a list box.

domain property and synchronized with its content. Similarly, an array of a favorite songs is organized into a collection displayed within a list box.

Elementary data typed objects form a significant part of any DSL model. The suitability of designed features for editing given objects directly depends on the experiences and design feeling of the developer itself. Thus the understanding of how these objects may be visualized is important to create projections, which allow the user to edit the language model adequately.

4.10.2 Projections of Complex Data Types

The language model does not contain only elementary data typed members. The complex ones usually dominate. Complex data types are primarily classes composed of another elementary or complex types. If these classes implement a projective interface, they become projective types, thus they are provided with their own projections. The language model must visualize all its members in a specific projection, let us denote them *parent projections*. If a parent projection is about to visualize a projective typed member, it does not know how to do it, since this knowledge belongs directly to this member. The way, how it can deal with that is to create a placeholder in a form of projective object control and binds the corresponding member to it. Projective object control will handle the projections of bound object on its own.

The projection of complex types is illustrated on the *Figure 4.27*. A data type called Mp3 offers its own projection. The placeholder in a form of projective object control is placed to the parent projection and bound into the *FavouriteMp3* member of a language model.

Separated projections of complex types allow hierarchical breakdown of the projections. Some components of the model can be then presented independently

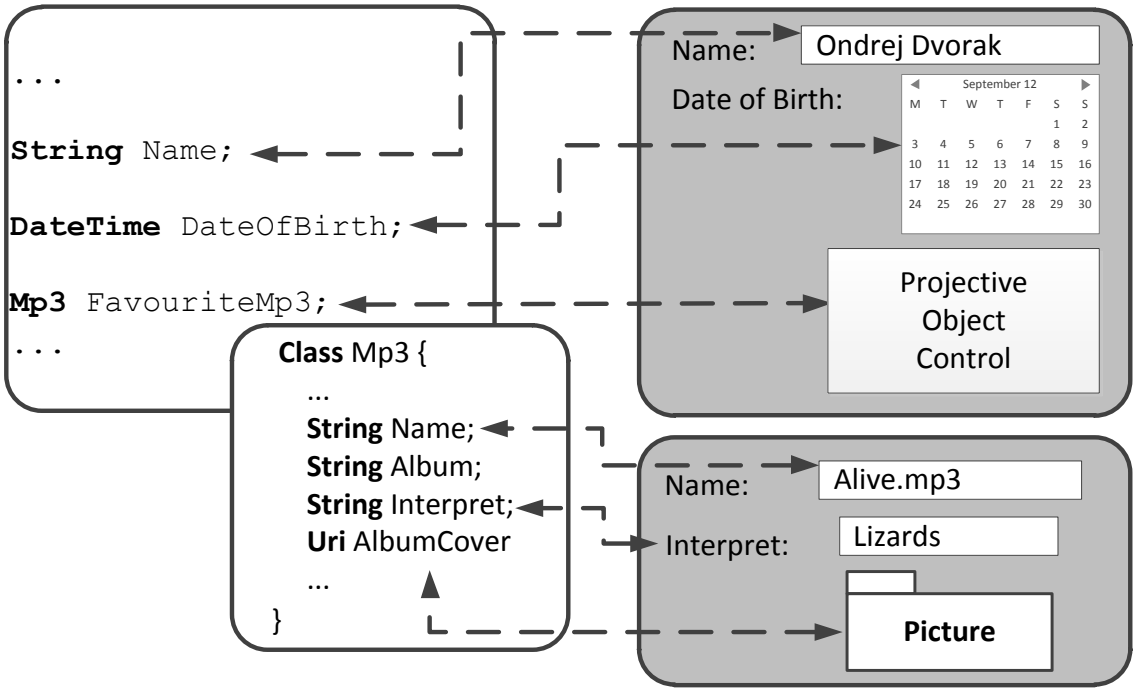


Figure 4.27: Complex projection includes a projective object control to visualize its complex member of Mp3 type.

regardless of the appearance of other parts of the model. Practically, this means that for example a piece of code can be displayed in a graph or in a table, despite the rest of the code is displayed textually. It is certainly one of the main advantages of projectional editors over conventional text editors.

4.10.3 Validations of Projections

Time to time we encounter a need of notifying the user that something is going wrong in a projection. For instance if a language construct that is displayed in a projection does not match the syntactic rules, it must be marked significantly. Since we analyzed that a model represents an abstract syntax of a language only. Therefore, the validation of an element syntax must be delegated to the developer that is responsible for a creation of a validation method on its own.

We mainly have to provide a clear way to tie the projection with a convenient method to validate it. The exact way of validations directly depends on the technical realization of projections, thus we will not pay an attention to this topic again in the analysis. The aim of this section is only to notice that validations of projections should be considered during the design of a projectional editor.

4.10.4 Sharing of Projections

We evaluated that a DSL can be provided with projections of various complexity, from the simple projections (like text box, combo box, etc.) to the complex projections tailored to the specific needs of a language. We said that it is likely to be useful to provide a factory method for a creation of those elementary projections since these are used quite often. The question is, if it makes a sense to implement a factory method for the complex projections as well. This would be a kind of the projections sharing.

Although an unlimited number of DSL exists and each language has a unique structure, similar characteristics can be found.

For example, most of DSL contain language constructs repeatable infinitely. On the level of model, these are usually organized in a collection. This collection is flexible, we can add new elements or delete existing ones appropriately. It is reasonable to design a generic projection that can present a collection of objects conveniently, for instance, in an extendable vertical list. Such a projection could be used across many domain-specific languages.

More examples of general projections can be found. However, when we are about to design a projection of a specific language construct, we should consider, whether known general projection does not cover our needs as well. If none of the existing projections is suitable for us, we should evaluate, whether it is handy to design the projection generally to tie it up with a given language. In that case, a projection should be kept separately to be used by another languages as well.

4.11 Analysis Conclusion

At the beginning of this chapter, we made a decomposition of an overall problem represented by the goals revisited in the *Chapter 3*. Few decision steps arose from the decomposition and we analyzed them one by one in the corresponding sections.

We concluded that a domain-specific language must be described by a XSD which implies that source codes written in that language will be XML files.

Out of the XSD, a language model is generated using a tool called *Xsd2Code*. This model is represented by a class, its members can be either of an elementary data type or of a class type that is generated as well. These generated classes correspond with the language constructs, they are partial and can be joined with another classes implementing the projective interface and thus become a projective types.

A head method of the projective interface returns so called projection containers grouping the projections into event-driven units. Here, the projections can skip between each other in accordance to events. An instance of the projective type is a

projective object and returns all available containers via the mentioned interface.

Projection containers of a specific class are associated with a projective object control that forms the fundamental graphical control in the whole design. It allows the user to switch between all defined containers. It is wrapped by a projectional editor component that handles its instantiation with available editor add-ins.

Editor add-ins hold the necessary information to support projections of custom DSL in the editor. Primarily define how an input source file can be deserialized into a projective object. A desired projectional editor can be generally any application (IDE, desktop editor, custom application) that is aware of add-ins with which it can supply a projectional editor component. Thus a proper distribution of add-ins to the hosting application must be done.

In our case a Visual Studio designer is proposed to meet the stated goals. It gathers add-ins deployed to MSVS by MEF and coordinates the communication with underlying projectional editor component and the rest of MSVS IDE. Basically an input file is deserialized into a proper projective object using the information distributed by an add-in. This projective object is bound into the root projective object control that initiates the creation of all source code projections.

To give a clear notion, about the potential types of projections and their requirements, the last section *Section 4.10* was devoted to the analysis of projections themselves. In the next *Chapter 5* an implementation of a project DSLPed that accompanies this thesis is described with respect to the analysis.

Chapter 5

DSLPed Implementation

Last few chapters revisited our goals and analyzed them under the terms of a given domain. None of them did not cover the difficulties in implementing the proposed solution. These are going to be discussed below. We will go through each analyzed decision step stated in *Section 4.1* and we will outline its possible implementation. This way, we will step by step constitute a tool called DSLPed by which this thesis is accompanied. Since analysis (see *Chapter 4*) resulted in the choice of technologies that are used in DSLPed, we will give use cases to meet the objective of this thesis under these technologies. At the end of this chapter we will evaluate them appropriately.

DSLPed[1] is discussed in this chapter. Its usage is demonstrated on an example in the *Chapter 6*. Thus after reading those two chapters, both the use cases will be satisfied. Since the basics of WPF that is a core technology of this thesis are provided in the *Section 5.4*, several technical steps require a deep understanding of this technology. The WPF insides and advanced development are unleashed in a book *Pro WPF in C# 2010: Windows Presentation Foundation in .NET 4* [16] and basically covers a fundamental knowledge required for the easy understanding of a projective component (see the *Section 4.6*) and a projectional editor (see the *Section 4.7*) implementation.

5.1 Use Cases

At first, we will give two use cases, which will lead us to design the architecture of DSLPed, as it will be described in this chapter. Each use case is based on a different point of view. Once the end-user point of view, second the developer point of view. Needless to clarify that in our context *end-user* refers to the one who is provided with a language and who would like to maintain its source codes via a

[1]The project is available on the http://dslped.codeplex.com and licensed under the Apache License of a version 2.0. To contact the author please use ondra.dvorak@email.cz.

projectional editor. *Developer* is meant by a man who implements the required projectional editor of a given language.

≫ ***Use Case: End-user Point of View***

There is a domain specific language and code files written in it. We would like to present and maintain a code using projections and allow easy switching between them. For this purpose we prefer to use an editor that is a part of Visual Studio.

≫ ***Use Case: Developer Point of View***

There are a domain specific languages described by XSD. We would like to create WPF based projections of their individual language constructs and implement a projectional editor that can deal with that information and load them via MEF. An editor, as well as this information is released in a form of Visual Studio extension.

It is clear, we need to implement a projectional editor that can be installed as a part of Visual Studio. If and only if an editor knows specific language and its projections, it can maintain the sources written in that domain-specific language. This knowledge must be kept in a suitable form, a unit which may be loaded by an editor. DSLPed is a concrete implementation of a projectional editor and a suitable tool for manipulating with the languages and creating desired units.

5.2 DSLPed Overview

DSLPed - *Domain Specific Language Projectional Editor*, a project supporting the projectional stuff implementation based on the thoughts which we came up with in the *Chapter 4*. It is not only an editor supporting the handling of domain-specific languages, it is a package (DSLPed *Package*) containing a generally applicable framework (DSLPed *Framework*) for the development of components requiring a kind of WPF projections. In addition to that, it provides an editor (DSLPed *Editor*) that can host various add-ins (DSLPed *Add-ins*) defining the possible projections of a specific DSL. The editor can be either integrated in the desktop application (DSLPed *Desktop Integration*), or as a part of the Visual Studio extensions in a form of a designer (DSLPed *IDE Integration*). DSLPed Framework contains elements that are tightly coupled with the projections (like the factories of typical projections), as well as general stuff associated with the editor itself. Since several of these stuff are too universal to be used in the context of projectional editors only, these are kept separately in common libraries (DSLPed *Common*) and may be used in a context of any WPF application. The core structure of DSLPed is on the *Figure 5.1*.

We will go through all the key parts of DSLPed within the subsequent subsections and we will introduce them briefly. Each of them will be described deeply later on. To have a notion how the source code, where these parts are implemented looks like, we will first describe the structure of DSLPed solution in accordance with its sources.

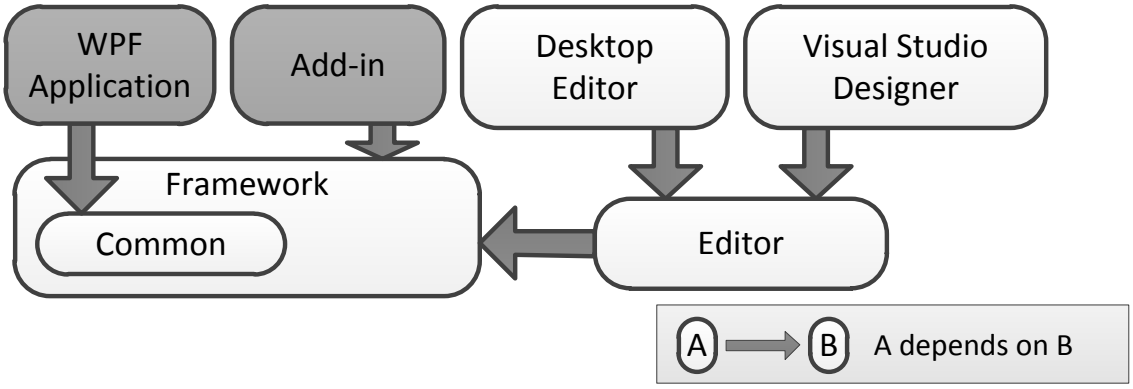


Figure 5.1: Libraries organized into disjunctive, dependent groups (Framework, Common, ...). The shaded groups refer to the libraries developed using DSLPed.

5.2.1 Structure of the Developed Solution

DSLPed is a Visual Studio solution of various C# projects. It is organized into few solution folders, whose names roughly correspond to the parts mentioned before (DSLPed *editor*, DSLPed *framework*, etc.). Some of the projects are paired with a testing one, which has the same name plus the suffix *Test*. Testing projects group NUnit[2] tests covering the core functionality of DSLPed.

5.2.2 DSLPed Editor

DSLPed editor is an essential part of the design. It is an independent component that can recognize configured languages. It can open their source code files, edit them in specific projections and save them back to the file.

5.2.3 DSLPed Common

Since DSLPed Package contains helpers, .NET extension and utility methods, WPF convertors and controls that may be used in the context of any .NET application, those have been moved to separated libraries. The libraries can be referenced by any .NET application, regardless of whether the application handles projective content or not. These are the basic libraries, which do not depend on any other DSLPed

[2]Testing framework for .NET languages

package component and therefore their use does not require the entire package installation.

Beside the mentioned .NET components, the WPF controls library is also a part of the DSLPed Common. In fact it contains basic building blocks of elements intended to implement the projective controls themselves. In principle it is possible to use them to create custom projective controls. Since these elements laying the foundations of the entire design, they will be discussed in a separate section.

Why not to keep it together with the framework

Clear separation of the framework-specific stuff and general purpose elements do have a sense for the whole solution. Developers are not forced to link their applications with a large framework whose greater part of the functionality offered remains unused. Moreover, it is quite likely that there will be more developers interested in the core elements than in the framework. However, their feedback on the core parts functionality may have a valuable impact on the framework itself.

5.2.4 DSLPed Framework

DSLPed framework is a set of libraries implementing the fundamental elements for the development of DSLPed editor add-ins. Its main part is built of common projections, their factories, extension methods tightly coupled with the projections and API for a communication between add-ins and an editor.

5.2.5 DSLPed Visual Studio and Desktop Integration

As we discussed before, DSLPed editor is a kind of an independent component that may be hosted by another WPF component of application. Visual Studio and Desktop Integration are an example of such the applications.

Visual Studio Integration

It is a set of VSPackages and libraries containing MSVS templates, which are grouped into a DSLPed VSIX package. Its core component is a VS Designer, which main control directly hosts the editor and is responsible for the interaction between the editor and IDE. It is enriched with Item and Project templates for the easier creation of whole add-ins structures and their crucial components.

Desktop Integration

Desktop Integration is a lightweight implementation of the application that hosts an editor. It is not burdened with the complexity of an integration into the comprehensive IDE and it therefore becomes a suitable candidate for a testing of an Editor API. Moreover, it does not suffer from a long application startup, as the IDE does. Its manual testability is then relatively quick.

5.3 Model of a Language

This section implements a decision step D1 stated in *Section 4.1*, the analysis of D1 is in the *Section 4.2*.

We saw that several ideas of this solution rest on a language model. We discussed that a language model represents an abstract syntax of a language and it therefore covers mainly the relations between the language constructs. We stated that the model can be described by XSD and corresponding C# classes can be generated out of it using a specialized tool. The only thing missing is a deeper look on the model itself and a tool we can use to generate it, thus we will deal with it in this Section. First we will take a look on a tool known as *Xsd2Code*, which is aimed at generating the .NET classes out of the XSD, second we will demonstrate a model of a simple language called *Beep*. We will show how the model looks like and how it was created with the use of *Xsd2Code*.

5.3.1 Model Generator

Xsd2Code is a generator of .NET classes from XML schema. It is an open source project which is fully integrated in Visual Studio after it is installed. Generally nothing prevents us from using another similar tool, however, we found out *Xsd2Code* a suitable tool for us. If we ever decided to replace it by anything else, we should bear in mind that the generated classes should have the following aspects not to get into troubles.

- ***The generated class must be partial***

  This requirement is kind of tricky. It allows us to enhance a class, to create its additional part and implement any interface we are interested in. For instance, we can make a usual class a projective one that is capable of self-projections. Such a mechanism is described in more details later on.

- ***The generated class must implement INotifyPropertyChanged***

  INotifyPropertyChanged plays an important role in binding. A class that implements this interface may notify binding clients that a value of its property

has changed. The binding client is meant by a view element that is bound to any of the class properties.

≫ ***Generated class must be serializable***

The generated class represents a language model described by XSD. The source code of that language is either a class instance or a XML file. These are mutually convertible if and only if the class is serializable. Thus its instances can be stored as XML files.

As soon as the *Xsd2Code* is applied to the XSD defining the language, the corresponding model of a language in a form of .NET classes is created. The *Figure 5.2* demonstrates its use.

Figure 5.2: Language model generated out of the XSD using *Xsd2Code*

5.3.2 Beeps Language Model

Model of a language is nothing but few .NET classes. Their properties directly correspond with the elements presented in the XSD language definition. *Beeps* is a simple language designed for the illustration purposes only. It consists of one language construct - *Beep*. *Beep* is a command that executes a sound of a given frequency in Hertz. The program written in *Beeps* is just a sequence of *Beep* commands operating on various sound levels. A snapshot of a program written in *Beeps* is on the *Listing 5.1*. The sample of its model looks like on the *Listing 5.15*. Its XSD language definition out of which the model was generated is on the *Listing 5.3*.

```
// Execute beeps in the given sequence
Beeps {
  Beep(500);
  Beep(1000);
  ...
  Beep(100);
}
```

Listing 5.1: A snapshot of a program in *Beeps*

```
/// <summary>Root construct in Beeps model</summary>
public partial class Beeps {
    // Collection of beeps
    private List<Beep> beepsField;

    /// <summary>Gets or sets the collection of beeps</summary>
```

```
    public List<Beep> beeps {
        get {
            return beepsField;
        }
        set {
            beepsField = value;
            OnPropertyChanged("beep");
        }
    }
}

/// <summary>
/// Sub-construct in Beeps model
/// </summary>
public partial class Beep {
  ...
}
```

Listing 5.2: A sample of *Beeps* model

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://Beeps">
  <xs:element name="beeps">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="beep" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="frequence" type="xs:decimal"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 5.3: XSD *Beeps* definition

5.3.3 Model Enhancement

The model itself does not contain any information but the structure of all the language constructs and the relations between them. If we wanted to enhance a model of any other knowledge, which would be gathered by some interface method, we would need to do it manually.

The very first thought that springs to mind is to edit directly the generated code and just extend it with the desired method. This will unfortunately result in an unpleasant effect on our code just after we regenerate the model again. All the changes will be suddenly lost. We discussed that it is important that .NET classes generated out of the XSD are partial, and this is the time when this requirement comes into play.

If the class is partial, its definition can be split over two or more source files.

We can therefore create an additional file containing a class of the same name and implement the missing method in it. None of our changes will be lost after regeneration of the model since all of them have been created in a separated file. Both the parts of class will be combined when the application gets compiled.

Needless to say that such an approach may be applied to serve any requirements for model enhancements, when the direct changes of a model are undesirable. As it may be seen, the main knowledge, with which we need to extend the model, is a visual appearance of the model elements. The concept of partial classes makes the things easier and we will apply it in the *Section 5.5*.

5.4 Projections

This section implements a decision step D2 stated in the *Section 4.1*, the deeper analysis of D2 is in the *Section 4.3*.

We already discussed that a projection is a visual form of a certain language construct. In the *Chapter 4*, we stated, WPF is the most suitable technology in our context. Since we listed several benefits of WPF, we did not touch the actual implementation of WPF-based projections yet. This is a purpose of the current section. We will explain projections in WPF terms. To do it properly, we will pass the basics of WPF first.

5.4.1 WPF Introduction

Windows Presentation Foundation (WPF) is a Microsoft's graphical subsystem for Windows applications. Since many comprehensive books about WPF have been published, its deeper investigation and explanation is out of the scope of this work. Thus we will give a nutshell introduction, we will try to offer its bases, which are indispensable for anyone creating projections. These bases are covered by a book *Programming WPF*[20]. On the other hand, a book *WPF Programmer's Reference: Windows Presentation Foundation with C# 2010 and .NET 4* [19] may serve as a handbook. The knowledge gained by reading these two books is sufficient for the development of projections.

5.4.2 WPF Architecture

In this section, we will take our first look to an overall architecture of WPF. We will point out the things that the implementation of projections relies on.

Since projections are usually composed of common elements like *TextBoxes*, *Buttons* etc., we will show where to find them. WPF is based on a multilayered architecture. The core libraries which we interact with are spread in the highest level.

Those libraries, written in managed C# code are mainly the ones with whom we usually come into contact. The highest level is represented by *PresentationFramework.dll*, *PresentationCore.dll* and *WindowsBase.dll*, whose are usually referenced by our WPF projects. The underlying WPF is based on DirectX[3] and thus the translation of .NET objects into Direct3D[4] textures is done behind the scenes. The outline of a WPF architecture is on the *Figure 5.3*.

Figure 5.3: WPF architecture: Applications created using core libraries based on the Direct3D

Projections mostly revolve around a term WPF Control or in short a *control*. *WPF Control* is a basic building block of each WPF application. *Buttons*, *TextBoxes*, *ListBoxes*, *ListViews*, *CheckBoxes* - all of them are WPF Controls.

It is helpful to know at least the basic hierarchy of classes which the WPF Controls depend on. At the top of the hierarchy (see the *Figure 5.4*) a *DispatcherObject* takes its place. It is a very base class of the underlying specialized classes. From the projection point of view, the most interesting branch of a hierarchy is the one rooted in a Control class. General projections provided in the DSLPed framework inherit exactly this class, or the less specialized ones. In WPF terminology, these projections are represented by CustomControls. A simple aggregation of WPF Controls into a unit is called UserControl since a corresponding class usually inherits a UserControl class. *Subsection 5.4.6* is devoted to the characteristics of CustomControls and UserControls.

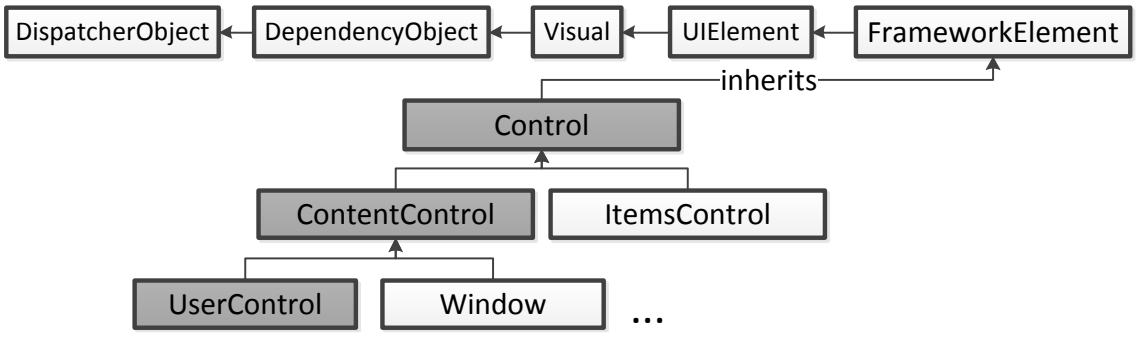


Figure 5.4: WPF class hierarchy. Projection-relevant elements are shaded.

In this section, we outlined a general WPF architecture and a structure of its core classes. We set a term WPF Control which the whole architecture relies on.

[3] A collection of Microsoft's libraries and API for the handling of multimedia tasks
[4] A sub-component of DirectX for a rendering of three dimensional graphics

We pointed out that projections are in fact equal to WPF Controls that are built using already existing controls.

5.4.3 XAML

Extensible Application Markup Language (XAML) [17] is a markup language used to define WPF user interfaces. Mastering its syntax is a prerequisite for making projections, since their visual layer is written in it. In this section, we will try to cover its key concepts and learn the basic rules when writing XAML code. First we will introduce XAML itself, we will briefly discuss benefits gained by using it. Afterwards, we will describe an elementary piece of XAML code and its tight cooperation with a code-behind (see the *Subsubsection 5.4.3.4*).

5.4.3.1 Introduction

XAML is quite a handy language. Without writing a single line of C# code, we can design WPF windows and user controls with the use of features like data binding, triggers or templates. XAML is a XML-based language, which we can either write by hand, or more likely generate using a dedicated designer tool, for instance with the use of Expression Blend[5] or even using an integrated MSVS designer. In a WPF concept, a window or a user control is serialized into a set of XAML tags, out of which the .NET objects representing the user interface are generated. The generated objects are stored in temporary files with an extension *.g.cs (see the *Figure 5.5*).

Figure 5.5: Files of an extension *.g.cs containing .NET objects generated out of the XAML

XAML is thus not required by WPF. We can create .NET objects of the user interface directly. However, this would tie the presentation layer of the application with its core logic. We would lose the possibility to leave the creation of graphics to designers and afterwards painlessly merge it with a core logic of the application implemented by programmers. Thus XAML is not a mandatory tool for the creation of WPF user interfaces and projections, nevertheless the creation is much easier with its use.

[5]Microsoft's tool for designing graphical interfaces

5.4.3.2 Ground Rules

With respect to the book [16], XAML has quite a straightforward standard that is clear after a few ground rules below are understood, we should keep them in mind during the design of projections as well. Their visual outline can be seen on the *Figure 5.6*.

- XAML is a XML-based language, therefore it contains various elements (XAML *elements*) and their attributes (XAML *attributes*).
- Every XAML element is mapped to an instance of a .NET class, while a name of the class is equal to the name of a XAML tag. For instance, a tag <TextBox> results in the instantiation of a *TextBox* class.
- The XAML elements can be nested, which corresponds to the visual appearance of a created user interface. For instance, a <`TextBox`> element nested in a <`Border`> element results in a creation of a border surrounding a *TextBox*.
- Classes correspond to the XAML elements, properties of the classes correspond to attributes of the elements, through which the properties are accessible in a XAML code.

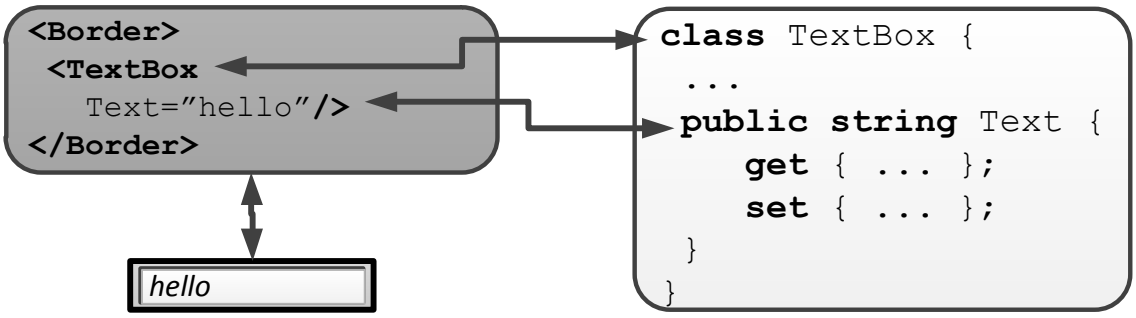


Figure 5.6: Ground rules outline: A suitable .NET class is generated out of the XAML that represents a visual control.

5.4.3.3 Blank XAML Control

We mentioned XAML or WPF several times before, but we did not touch the code yet. To have a better idea, how XAML language looks like, below on the *Listing 5.4*, there is an example of a blank UserControl. A code of the blank custom projection will look similar to that.

A document basically describes an empty user control. It consists of a <UserControl> element on the top and a nested <Grid> element. Each element in XAML can contain various attributes. These correspond with the properties of a class represented by an element. The most interesting one, placed in a root <UserControl >

element is an attribute *Class*. It refers to a code-behind class where event handlers can be implemented.

We will discuss the code-behind class in the next section. For now, it is sufficient to know how the XAML code looks like in general and to have a notion about a relationship of XAML elements, .NET objects and their visual appearance.

```
<UserControl x:Class="WpfControlLibrar1.UserControl1"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
    </Grid>
</UserControl>
```

Listing 5.4: Blank XAML code

5.4.3.4 Code-behind

Since several different events may be raised during the interactions within the user interface of a projection, these must be handled appropriately. A code-behind class was created to serve these purposes. Code-behind class is a class automatically generated at a compile time[6], while it can be supplied by the custom event handler methods. In accordance with design patterns, this class can be considered as a ViewModel in MVVM or as a Controller in the MVC pattern. The outline of the code-behind in a combination with data binding into a model is on the *Figure 5.7*. Data binding will be described below in this chapter.

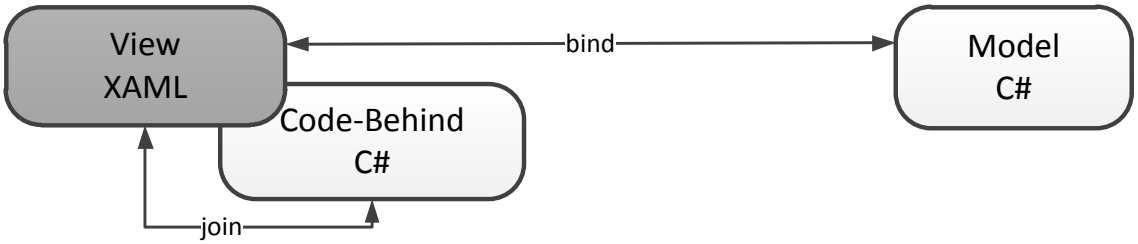


Figure 5.7: WPF Code-behind joined with the XAML and a XAML bound into a model

We already mentioned that a .NET class is generated out of a XAML code. In fact, this class is partial, thus its definition can be spread over several files. The code-behind class is partial as well and it even bears the same name as the generated class. During a compilation, both the classes are merged together behind the scenes.

[6]Code-behind usually represents the code of GUI that is placed in a separated class and allows us to separate the GUI and the logic behind it.

However, for us, it is important that a XAML code is provided with a code-behind class, where we can put the event handlers of our projections. A blank code-behind generated for the *Listing 5.4* is on the *Listing 5.5*.

```
namespace WpfControlLibrary {
    /// <summary>Interaction logic for UserControl1.xaml</summary>
    public partial class UserControl1 : UserControl
    {
        public UserControl1()
        {
            InitializeComponent();
        }
    }
}
```

Listing 5.5: Blank XAML code

5.4.4 WPF Binding

To tie the projections with a language model they present, we need a proper facility for it. WPF data binding is a desired tool we are looking for. In general, data binding is a relationship between *source* and *target* objects. Data fetched from a source object are used to set a target property of a WPF element. An overall goal of data binding is to separate user interface from the business logic. An object representing a business logic can be simply bound to a view. Its visual appearance is afterwards exclusively in the hands of a view logic. WPF data binding is quite a complex mechanism, since several comprehensive books about XAML and data binding were published, we will only focus on the key concepts of binding and we will consider, how it may be used in a context of projections.

5.4.4.1 Basic Concept

Data binding establishes a connection between a source and a target element. There are different types of bindings, like *OneWay*, *TwoWay*, or *OneTime*. Behavior of a specific binding instance is dependent on this settings. In general the changes that occur in a bound object are automatically reflected to a corresponding element and vice versa. For instance, if a <TextBox> element is bound into a string property of an underlying object, this property is set automatically, after the text inside a <TextBox> is changed. Coincidently, when the string property of this object is changed by a business logic, its value is automatically updated within the <TextBox> element. The binding between a XAML code and a corresponding C# property is illustrated on the *Figure 5.8*.

From the projections point of view, if a <TextBox> represents a value of a certain language element, the text in that <TextBox> is synchronized with the property representing the language construct on the level of model.

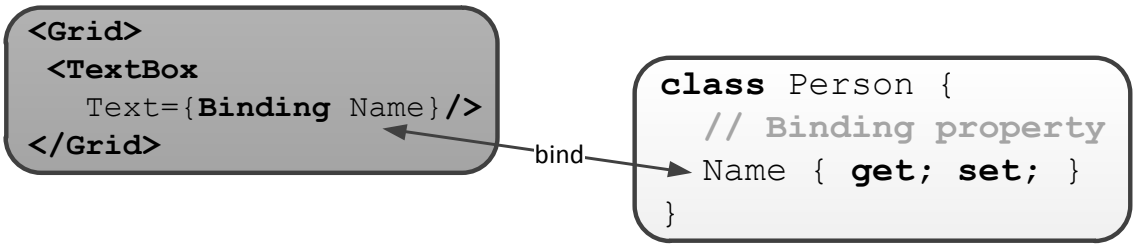


Figure 5.8: WPF Binding illustration: A text property of the *TextBox* is bound into a Name property of the class Person.

5.4.5 WPF Styles and Templates

Time to time, we have a need to provide a set of projections with a unique style (in terms of look and feel), or a need to customize a general projections with respect to a given DSL. WPF styles and templates are the essential features that are built to serve these purposes. These will be described in the following sections.

5.4.5.1 Styles

We noted, a XAML code is composed of elements and their attributes. A collection of attribute values that can be applied to an element is known as *Style*. Since an element is represented by a certain WPF Control class and its attribute refers to the property of this class, a style in fact sets the property values of a corresponding class instance. Needless to say that only *dependency properties* are affected by the style (read details about dependency properties in [16]). On the other hand, most of the properties of common controls are dependency properties, therefore we do not have to care about this limitation. This fact plays a role only during the design of CustomControls, which are discussed below. Style is useful in case we want to standardize some look or behavior. For instance if we want to provide a set of *TextBoxes* with a custom appearance (background, font color, etc.), we can simply create a style that defines the required look and apply it on given *TextBox* elements.

Furthermore, a style may be conditional by so called *triggers*, when the style of a control depends on the value of a defined property.

In the context of projections, style may be applied to an overall look of certain projections. For instance, if we design a custom projection that contains brackets and we provide a property which refers to the color of these brackets. We can simply create a style to color the brackets specifically at once. Custom projections are discussed in the *Subsection 5.4.7*.

5.4.5.2 Templates

a a WPF *template* defines a visual appearance of the control. Each control is provided with a default template, which is usually distributed together with it. By setting the template, we can completely replace the visual appearance of the control. For instance, we can transform the appearance of a *Button* to a *TextBox*. Since the templates deserve more explanation, refer for the details to [16]. For us, it is mainly important that when we are about to create a new custom projection in a form of CustomControl (see the *Subsection 5.4.6*), we have to provide it with a default template.

5.4.6 WPF CustomControl Versus UserControl

If we ever encounter a need to create an independent graphical component with its own core logic, in WPF we usually have two options. Either implement a CustomControl or design a UserControl. Since we have to consider them before implementing a projection, in this section we will briefly describe both and we will touch the difference between them and their suitability for projections.

A *UserControl* is an aggregation of existing controls into a reusable component. It consists of a XAML code and a corresponding code behind. An easy design of UserControl can be achieved using a designer integrated in MSVS. Unlike the CustomControl, a template or a style cannot be applied to a UserControl. From the implementation point of view, it is a class whose parent is a UserControl.

A *CustomControl* is not just a simple composition of controls. It inherits already existing control and extends it with additional features. It is always provided with a default look, which is stored in the *Themes/Generic.xaml* file. The main difference from a UserControl is its ability to work with templates and styles. Its visual appearance can be changed when a template or style is applied on the CustomControl.

From the above it follows that a WPF Control, whose behavior and look depends on an overall template should be created as a CustomControl. If no such a need is required, it is easier to aggregate it with the use of already exiting controls. The same point of view should be used when designing projections.

5.4.7 WPF Control as a Projection

In the previous sections, we built a knowledge portfolio, which should be sufficient to understand the explanation of projections in WPF terms. When creating a projection of a language construct, we usually have two options. Either use one of the *general projections*, which are already implemented in the DSLPed, or create own *custom projection*. A custom projection is either a UserControl or a CustomControl

that is bound into a model of the element using WPF binding. If we have to decide between a UserControl and a CustomControl to represent the projection, we have to consider whether we need to apply a style to the projection and whether the general CustomControl is likely to be useful. If a projection tends to be reusable in several contexts, it is reasonable to implement it in a form of CustomControl and design a template, which the projection is shipped with. However, since a CustomControl is a more demanding way, we should pick a UserControl for the projections that are disposable and that do not need to by styled.

5.5 Projective Object

This section implements a decision step D3 stated in *Section 4.1*, the deeper analysis of D3 is in the *Section 4.4*, where we scrutinized a projective object.

Each class participated in a language model may implement a projective interface, which makes it so called projective class. Projective object is an instance of a projective class. We discussed that a projective interface itself is a shortcoming, since we force a standalone class to implement an interface. However, a projective interface is a kind of a model enhancement and we found a workaround to enhance model by applying a concept of partial classes (see *Subsection 5.3.3*). Therefore, we can keep the definition of projections in a separated partial class and thereby not to interfere with the model itself.

The model itself does not contain any information about the appearance of the language constructs. The only information it knows is the structure of all the language constructs and the relations between them. Therefore, we have to enhance the model of a new knowledge - a visual appearance (projection). We are already familiar with the language model creation, as well as the implementation of projections. Thus in this section, we will show how to put these things together and implement a projective object for the given model and its projections.

In the section *Section 4.4*, we outlined a possible look of the projective interface and we concluded that it must contain a method gathering the available projections.

However, we faced a problem of event-driven projections and we analyzed its solution in the *Section 4.5*. This solution modified the simple projective interface and led us to a concept of projection containers, when the projections are grouped into isolated units with their own internal workflow. Next *Section 5.6* is devoted to the implementation of projection containers. Therefore, the aim of this section is relatively modest. We will just present a projective interface, how it looks like in DSLPed and we will slightly comment it. On an example, we will show how a specific language construct may be enhanced of a projections by implementing this interface. We will stay away from the details of projectional containers since these

are described in the next section.

5.5.1 Projective Interface in DSLPed

Projective interface in DSLPED looks like on the *Listing 5.6*.

```
public interface IProjective {
    /// <summary>Get available projection containers</summary>
    List<IProjectionContainer> GetProjectionContainers();
}
```

Listing 5.6: Projective interface

It contains the only method *GetProjectionContainers()* which returns all available groups of projections defined for the language construct represented by the class that implements this interface.

5.5.2 Beeps Model Enhancement

Now, let us go back to the model of a language *Beeps*, which have presented in the *Subsection 5.3.2*. It contains two language constructs, *Beeps* that is a root one and a *Beep* construct. *Beep* is in fact a sub-model which is represented by a class called *Beep*. To give this class a visual appearance, we need to promote it to a projective class, thus implement a projective interface. Since we do not want to touch the class itself by implementing a required interface method, we will create a new file, where we implement another part of the *Beep* class (see *Listing 5.7*).

```
public partial class Beep : IProjective {
    /// <summary>Definitions of projections grouped into projection containers</summary>
    public List<IProjectionContainer> GetProjectionContainers() {
        ...
    }
}
```

Listing 5.7: *Beeps* model enhancement

This is just enough to make a *Beep* class *self-projective*. In a compilation time, both parts of a *Beep* class will be merged. Later we will show how DSLPED controls interact with such class instances and how they present the available projection containers in the projectional editor.

5.6 Projection Container

This section implements a decision step D4 stated in *Section 4.1*, the deeper analysis of D4 is in the *Section 4.5*, where we took a look on the event-driven projections.

In the analysis of event-driven projections we focused on a goal G4a, which we stated in the *Chapter 3*. G4a requires enabling hops between projections based on events.

We analyzed that the hops should be realized inside isolated groups of projections, so called projection containers. Projective interface was introduced as an interface whose method returns a collection of such projection containers. At any moment one of the projection is active and my hop to another under specific circumstances associated with events. The circumstance is defined by an active projection and an event. Thus an information of the active projection, event and a the next projection to hop is sufficient to replace the active projection with another. Let us call this information a *transition*. A set of transitions in fact forms a workflow inside a container. Projections inside a container are not aware of each other. These can hardly ask for another projection to hop to since they do not know another name of projection but their own. On the other hand, a container knows all its projections and can define transitions among them easily.

In DSLPed, projection container is provided with an interface, which we will introduce in this section. An interface basically covers registration of projections and transitions in a container. We will comment all its members and we will show how the workflow among projections is defined. We will introduce a WPF control called `ProjectionContainerControl` that is bound into an object implementing the given interface. It presents the registered projections and switches them with regard to the workflow.

5.6.1 Projection Container Interface

A projection container interface is aimed at handling projections and transitions inside a container. The registration of projections and transitions must be provided by any class implementing this interface. On the *Listing 5.14* an interface is shown as it is defined in DSLPed.

```
public interface IProjectionContainer {
    /// <summary>Projections in the container</summary>
    ReadOnlyCollection<IProjection> Projections { get; }

    /// <summary>Default projection of the container</summary>
    IProjection DefaultProjection { get; }

    /// <summary>Name of the container</summary>
    string Name { get; set; }

    /// <summary>Image representing this container</summary>
    ImageSource Image { get; }

    /// <summary>Transitions between all projections</summary>
    ReadOnlyCollection<Transition> Transitions { get; }
```

```
    /// <summary>Register new projection in a container</summary>
    IProjectionContainer RegisterProjection(string projectionName, IProjection projection);

    /// <summary>Get projection registered under a given name</summary>
    IProjection GetProjection(string projectionName);

    /// <summary>Register a transition between two projections</summary>
    IProjectionContainer RegisterTransition(
       RoutedEvent routedEvent, string sourceProjection, string targetProjection);
}
```

Listing 5.8: Projection container interface

- ***Projections*** property is a collection of all projections that are grouped into a container. These may hop between each other in dependence on routed events.
- ***DefaultProjection*** is a reference to the initial projection. When we decide to interact with this group, this one will be shown at first.
- ***Name*** property defines the name of a container. The name should be representative enough to characterize the entire group of projections.
- ***Image*** is a picture characterizing the group.
- ***Transitions*** is a collection of transitions between individual projections. Transition contains a source projection, a target projection and an event that causes the hop from a source projection to the target one.
- ***RegisterProjection****(string projectionName, IProjection projection)* registers a new projection inside a container.
- ***RegisterTransition****(RoutedEvent routedEvent, string sourceProjection, string targetProjection)* registers a new transition between two projections identified by their name.
- ***GetProjection****(string projectionName)* returns a projection of a given name.

5.6.2 Registration of Projections Inside a Container

Any container that implements the interface mentioned above must be filled with projections. This is done using a *RegisterProjection()* method. In a *Subsection 5.3.2*, we introduced a simple Beep language. A language consists of a sequence of *Beep* commands that are kept in a property *beeps*. Here we will register a simple projection of a root *Beep* construct where its commands are displayed in a vertical order (see the *Listing 5.9*).

A new container is instantiated and vertical projection is registered for it. In this case a vertical projection is created via a factory method that is a part of DSLPed framework (see *Subsection 5.2.4*). It is bound into a *beeps* property of a language model. It means that in this projection, the *Beep* commands will be displayed one by one in a vertical layout.

```
new ProjectionContainer("SimpleContainer")
        .RegisterProjection("BeepsVertical",
                CollectionProjectionFactory.GetVerticalProjection<Beep>(beeps));
```

Listing 5.9: Registration of projection

A *Figure 5.9* outlines what is happening behind the scenes. An instance of a projection container is created and a *BeepsVertical* projection that is bound into a *beeps* property is added to the projections kept inside the container.

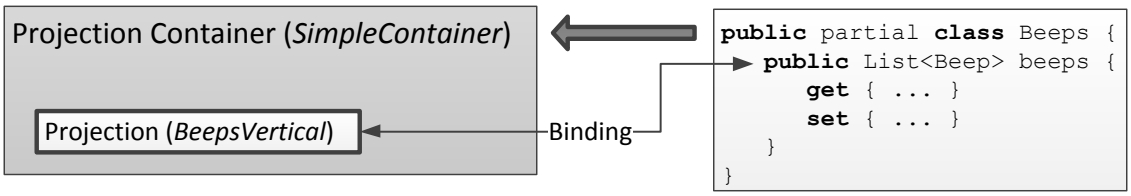


Figure 5.9: Projection container of a *Beeps* language construct represented by a class

5.6.3 Registration of Transitions Inside a Container

If there is more than one projection registered inside a container, we can define hops between them based on the events raised inside projections. These hops are so called *Transitions* and a kind of a workflow arises from them. Let us consider we have a WPF LinkedListControl, which is an ItemsControl that displays its items as a linked list. A control may raise an event GotFocus, when it receives a focus. On the *Listing 5.10*, a *BeepsGraphical* projection is created with the use of LinkedListControl and added into a container. A transition between a projection *BeepsGraphical* and *BeepsVertical* is required when the LinkedListControl. GotFocusEvent is raised inside the *BeepsGraphical*, thus a corresponding transition is registered. In other words, if a projection *BeepsGraphical* is presented and an event GotFocus is raised, we require to replace the *BeepsGraphical* projection with a *BeepsVertical* projection.

```
new ProjectionContainer("SimpleContainer")
        .RegisterProjection("BeepsVertical",
             CollectionProjectionFactory.GetVerticalProjection<Beep>(beeps))
        .RegisterProjection("BeepsGraphical", new LinkedListControl(beeps))
        .RegisterTransition(BeepsGraphicalControl.GotFocusEvent,
                            "BeepsGraphical",
                            "BeepsVertical");
```

Listing 5.10: Registration of transition

The *Figure 5.10* outlines a registration of new projection and a corresponding transition between the *BeepsGraphical* and the *BeepsVertical* projection. New instance of a projection that is registered under the name *BeepsGraphical* is created

and put inside a container. To tie the two existing projections together, an instance of a Transition is injected to the container via the *RegisterTransition* method.

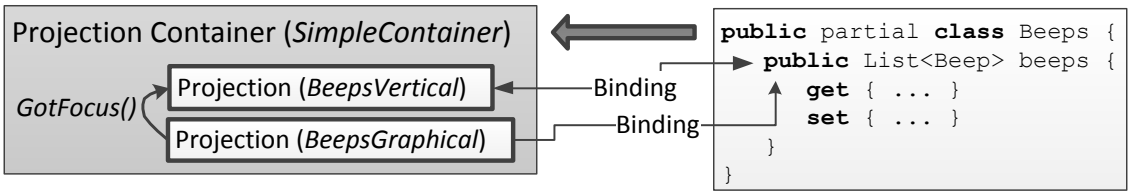


Figure 5.10: Transitions of projections inside a projection container. Two projections are bound into the same property of a class. The *BeepsVertical* is appeared just after the *BeepsGraphical* got a focus.

Needless to clarify that a projection container is a class that implements the projection container interface. The instance of this class is just a definition of some independent unit. On the example above, a projection container keeps three objects. Two instances implementing an IProjection and the instance of a Transition class. Projections themselves (as visual elements) are encapsulated in the IProjection objects. Therefore, we need a graphical WPF control that displays the projections defined in a container and switch them with respect to the workflow described by transitions. This control is a ProjectionContainerControl and we will discuss it in the next section.

5.6.4 ProjectionContainerControl

In the previous sections, we have introduced a projection container. Any class that implements the required interface can be seen as a projection container. It simply groups defined projections together and interconnects them using transitions. A set of projection container objects is returned by a projective interface (see *Subsection 5.5.1*). However, these objects are the definitions only. The corresponding projections must be visualized and their lifetime must be maintained by a specific authority. In DSLPed, a WPF control called ProjectionContainerControl is designed to handle it. Since a language construct can be provided with several projection containers, a WPF control that enables to switch among them must be created as well. A `ProjectiveControl` is designed in DSLPed to do this job. It is discussed in the next section.

5.6.4.1 ProjectionContainerControl as a Selector

ProjectionContainerControl is a Selector, which is an ItemsControl's derivate. It means that a collection of specific objects can be bound into it via an *ItemsSource* dependency property and the currently selected item is accessible via *SelectedItem* property.

In DSLPed, a ProjectionContainerControl is a general control provided by the DSLPed common package (*Subsection 5.2.3*). Its items are ProjectionControl objects that wraps any WPF control. An additional dependency property *Transitions* refers to a collection of TransitionControl elements.

Below on the *Listing 5.11*, a snapshot of XAML code shows the general implementation of a projection container which we defined earlier in the *Listing 5.10*. We will never write this piece of code manually, it is created behind the scenes by DSLPed. However, if we wanted to have event-driven WPF controls in any WPF application, we would do it in a similar way. It means, nothing prevents us from using ProjectionContainerControl in a general way to handle WPF controls. In our case, the projections are provided by a partial class of our model, we have to convert them appropriately and instantiate ProjectionContainerControl in a time when the corresponding language construct is about to be visualized.

```
<dsl:ProjectionContainerControl Title="SimpleContainer">
   <dsl:ProjectionContainerControl.Transitions>
      <dsl:TransitionControl FromProjection="BeepsGrahical"
                             ToProjection="BeepsVertical"
                             Event="GotFocus">
      </dsl:TransitionControl>
   </dsl:ProjectionContainerControl.Transitions>
   <dsl:ProjectionControl Name="BeepsVertical">
      <VerticalProjectionControl ItemsSource="{Binding ...}" />
   </dsl:ProjectionControl>
   <dsl:ProjectionControl Name="BeepsGraphical">
      <LinkedListControl ItemsSource="{Binding ...}"/>
   </dsl:ProjectionControl>
</dsl:ProjectionContainerControl>
```

Listing 5.11: Projection container definition in XAML

5.6.5 Achievement of the Goal G4a

In the section *Section 5.6*, we solved the decision step D4 that was stated in *Section 4.1*. Along with that we fulfilled the overall goal G4a revisited at the beginning of this thesis, in the *Chapter 3*. The definitions of projections are kept in so called projection container, where the required workflow is formed from event-driven transitions between projections. To allow hops between defined projections, we designed a ProjectionContainerControl that presents one projections with respect to their workflow inside a container.

5.7 Projective Control

In the *Section 4.1* we came across the need of having the component that manages the projections of a given language construct and we stated a corresponding decision

step D5. We analyzed that step in the *Section 4.6* and concluded that a projective control is responsible for swapping projection containers. Since the only one projection can be active (presented to the user) in a period of time, the projective control in fact swaps the active projections of two different containers.

On the other hand, we discussed that to achieve a certain level of generality, a projective control should not know anything about projections or containers. It should be able to handle any visual components regardless if these are projections or not. The component aware of specifics of projections should wrap the projective control appropriately. We called it Projective Object Control.

In this section we will take a look on the Projective Control and Projective Object Control as they are implemented in DSLPed.

5.7.1 ProjectiveControl

In DSLPed, projective control was implemented as a custom control WPF control named unsurprisingly `ProjectiveControl`. Its implementation is stored in the DSLPed common package (*Subsection 5.2.3*). It can be used in any WPF application to handle any WPF content. It inherits from an ItemsControl's derivate called Selector, thus its content is composed of a set of items and the selected one is presented in a convenient way. A `ProjectiveControl` in a hierarchy of common .NET objects is shown on the *Figure 5.11*.

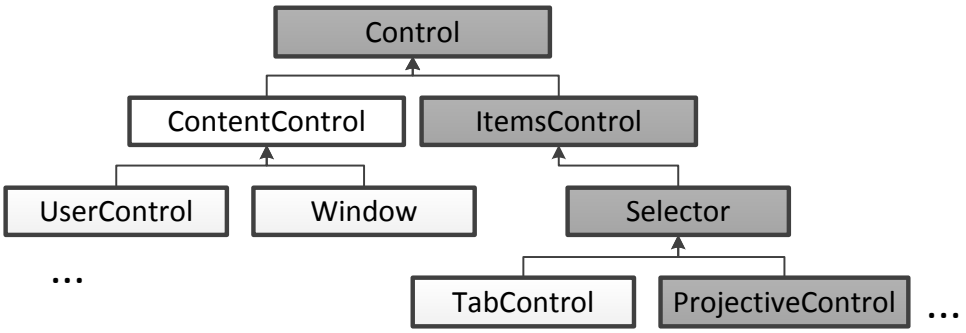


Figure 5.11: ProjectiveControl in a hierarchy of .NET objects

For instance, a `TabControl` is a `Selector`, whose items are arranged in tabs. Exactly one tab is active and refers to a *SelectedItem* property.

`ProjectiveControl` behaves in a similar way. Its items are WPF controls. Exactly one item is selected and picked to be displayed. It does not care what items it displays, if it is a common WPF control or a specialized projection container. If the item is a projection container, the responsibility of displaying underlying projections is delegated to the container itself. Similarly if an item to display is a `TabControl` a `TabControl` is responsible for displaying the content of a selected tab.

Thus the only job of a `ProjectiveControl` is to let the user chose from

available items and present the selected one properly. On the *Listing 5.12*, a `ProjectiveControl` is used in a XAML code in the completely general way. Its items are Labels of specific names.

```
<dsl:ProjectiveControl>
    <Label x:Name="HelloProjection">Hello Projection</Label>
    <Label x:Name="ByeProjection">Bye Projection</Label>
</dsl:ProjectiveControl>
```

Listing 5.12: ProjectiveControl in XAML

5.7.1.1 ContextMenu for Selections

We outlined that a `ProjectiveControl` is provided with a set of WPF Controls and enables to swap among them. In the DSLPed solution a context menu was chosen to present all available controls. A context menu is customized and contains a ListView where the user can pick one control to present. The *Figure 5.12* shows the context menu provided by a `ProjectiveControl` to change its items.

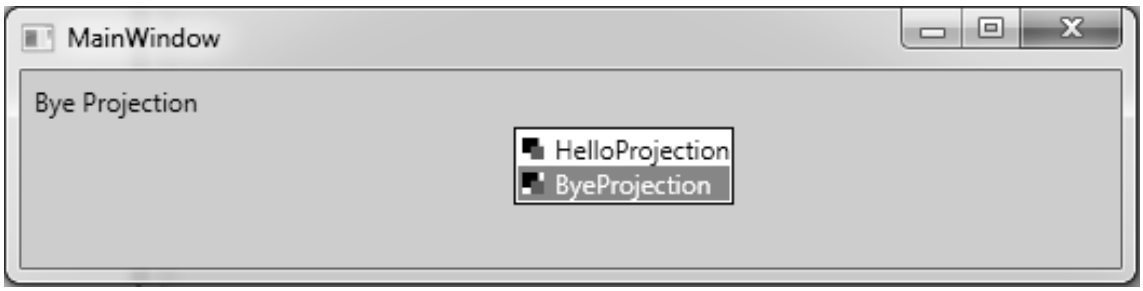


Figure 5.12: Context menu for changing the currently selected projection

5.7.2 ProjectiveObjectUserControl

Since `ProjectiveControl` is purely general component and we have to tie it up with projections coming from a projective object, we have to implement a control that adapts the projective object to the `ProjectiveControl`.

`ProjectiveObjectUserControl` does this job in DSLPed, where it can be found in the DSLPed framework package (*Subsection 5.2.4*). Its main goal is to adapt definitions of containers provided by a projective object and present them as items in `ProjectiveControl`.

In fact the adaptation process instantiates a `ProjectiveControl` class. If we implemented it in a XAML code, we would do that similarly to the *Listing 5.13*. However, this object is created behind the scenes by a `ProjectiveObjectUserControl`. `ProjectiveContainerControl` is created for each container definition returned by the Projective Object Interface. These objects are afterwards shipped as a collection to the `ProjectiveControl` that handles their presentations.

```
<dsl:ProjectiveControl>
   <dsl:ProjectionContainerControl Title="FirstContainer">
      <dsl:ProjectionContainerControl.Transitions>
         <dsl:TransitionControl FromProjection="HelloProjection" Event="MouseMove"/>
         <dsl:TransitionControl FromProjection="ByeProjection"/>
      </dsl:ProjectionContainerControl.Transitions>
      <dsl:ProjectionControl Name="HelloProjection">
          <Label>Hello Projection</Label>
      </dsl:ProjectionControl>
      <dsl:ProjectionControl Name="ByeProjection">
           <Label>Bye Projection</Label>
      </dsl:ProjectionControl>
   </dsl:ProjectionContainerControl>
   <dsl:ProjectionContainerControl Title="SecondContainer">
   ...
</dsl:ProjectiveControl>
```

Listing 5.13: ProjectiveControl with ProjectionContainers in XAML

5.7.2.1 Adaptation Process

We noted that `ProjectiveObjectUserControl` is an adapter of DSLPed-specific projective object to the common `ProjectiveControl`. The process of adaptation is more or less a technical issue, which is solved by a `ProjectiveControlAdapter` class. This class is instantiated for a given projective object, grabs the definitions of its containers and generates a `ProjectionContainerControl` for each. At the end it injects all the generated controls as items of a `ProjectiveControl`. The *Figure 5.13* demonstrates the flow of adaptation process.

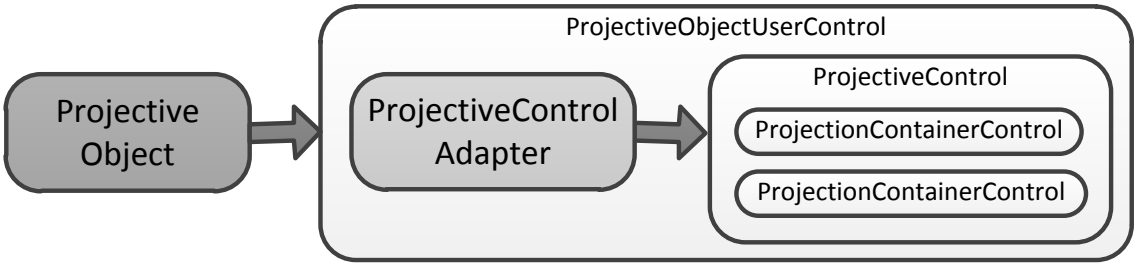


Figure 5.13: ProjectiveObjectUserControl adapts the bound Projective Object into the nested ProjectiveControl

5.7.3 Achievement of the Goal G4b

In the section *Section 5.7*, we implemented a projective component as it was analyzed in the *Section 4.6*. We demonstrated that a universal implementation of a corresponding `ProjectiveControl` gives us the possibility of using the control in distinct contexts. One of them is the DSLPed, where we can use it for the swapping between certain active projections, thereby fulfilling the overall goal G4b established in *Chapter 3*.

5.8 DSLPed Editor

In the *Section 4.1*, we were routed to the decision step D6, which claimed that a composition of projective components leads to an editor. We proved that statement in the *Section 4.7*, where the projectional editor was seen as an isolated component wrapped by a specialized IDE class, desktop application or in general by any WPF control. To ease the wrapping, the isolated editor must provide a clear interface.

In DSLPed solution, such an isolated component is represented by a WPF control called `DSLPedEditor` which implements an interface `IDSLPedEditor`. These will be discussed in this section. However, DSLPed editor is an essential component of the whole solution, therefore it deserves an introduction before we proceed with the technical aspects of its design.

5.8.1 Isolated Component

DSLPed editor is a component, whose only and main job is to open a source code file of a domain-specific language, present it using defined projections and save it back to the file system after the user edits it. The DSLPed distributes an editor encapsulated in a desktop application, as well as in MSVS designer, but nothing prevents us from using it as component of any WPF Control.

The main reason why it is kept independently is that a desktop editor and MSVS designer may vary in the file handling and interaction with the rest of application or IDE. In one case, the main desktop application menu is used for finding the file, whereas in another case, one of the Visual Studio facilities is used. Either way, the actual process of file opening, loading and saving will remain the same after the name of a file is gathered.

Therefore, the DSLPed editor provides an interface `IDslPedEditor` to hide the complexity of the GUI (IDE, desktop application, ...) from the editor itself. This well-defined interface enables us to use the editor in a context of any WPF application or IDE implementing it and allows us to reach a kind of an editor isolation.

5.8.2 Editor Interface

The crucial parts of an interface via which the communication with DSLPed editor is established are below. The interface is not intentionally complete since another members might lead to a confusion till we explain the extensibility of an editor.

```
public interface IDslPedEditor {
    /// <summary>Loads a file of a given name into a projective object</summary>
    object LoadFile(string fileName);
```

```
    /// <summary>Saves the loaded projective object into a file of a given name</summary>
    bool SaveFile(string fileName);

    /// <summary>Indicates whether a projective object was changed</summary>
    bool IsDirty { get; set; }
    ...
}
```

Listing 5.14: Projection container interface

- ***LoadFile(string fileName)*** opens a file of a given name and tries to deserialize it to a suitable projective object.
- ***SaveFile(string fileName*** tries to serialize the loaded projective object into a file of a given name.
- ***IsDirty*** property defines whether the loaded projective object was changed after it was loaded.

An interface can be used to coordinate the flow inside DSLPed editor. In the *Subsection 5.8.4* we will briefly describe this flow.

5.8.3 Editor View

Editor is represented by a common WPF control. The layout of its view is organized as shown on a picture *Figure 5.14*.

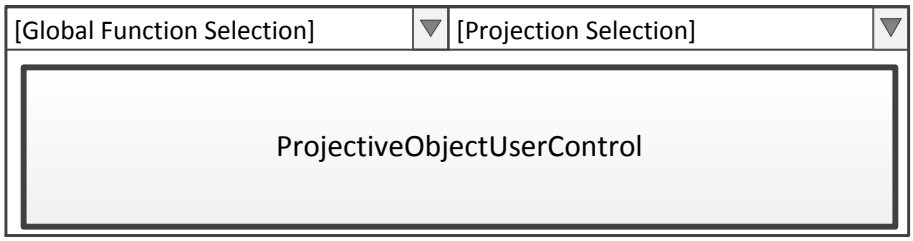


Figure 5.14: DSLPed editor layout organization

`ProjectiveObjectUserControl` takes the place in the main area of the view. Root projective object of the language that is maintained by an editor is bound into this control, which displays its projections. Within the projections, there can be `ProjectiveObjectUserControls` of another language constructs, each of them can be interactively selected by the user using a mouse click, while a *Shift* key is pressed.

Upper left corner is reserved for the display of all functions applicable globally on all `ProjectiveControls` that are currently viewed within the main area. For instance a function that notifies all `ProjectiveControls` to show names of projection containers they hold.

As we clarified, an editor is composed of a number of `ProjectiveControls`. Each of them is provided with a set of projection containers which we can switch. To

invoke a switch, we have to select a `ProjectiveControl` and open a context menu over it. The available containers to switch are listed in the context menu. The same result can be achieved using a menu in the upper right corner that is synchronized with the currently selected `ProjectiveControl` and lists all its containers.

5.8.4 Flow Inside an Editor

We already presented DSLPed editor as an independent component responsible for opening file, maintaining the code and saving the file back to a file system. It does not care how the file name is gathered (this task is delegated to a higher level), it just handles it via the interface `IDslPedEditor`. In this section, we will describe what is happening, when the higher level component asks for opening or saving a given file (see the *Figure 5.15* for the illustration).

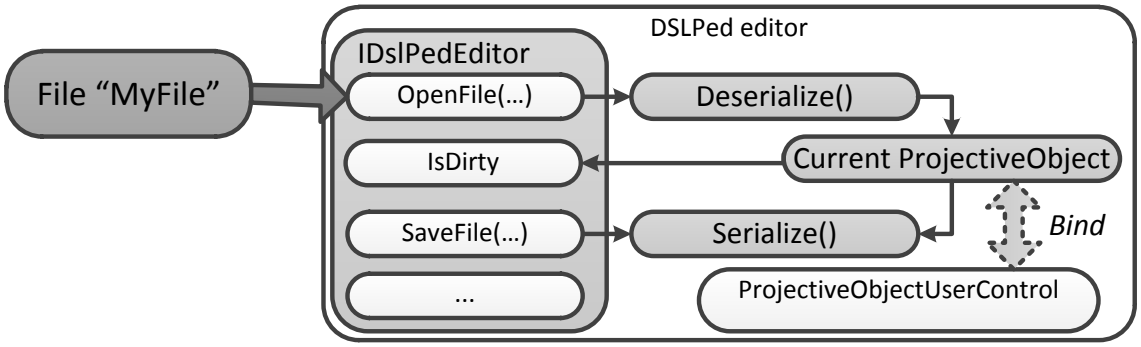


Figure 5.15: Flow inside the DSLPed editor

1. A higher level component (IDE, Desktop application, ...) asks for the opening of a source file named "MyFile".
2. Appropriate interface method `OpenFile()` is executed.
3. An editor opens a file and deserializes it into an object of a proper projective type.
4. An object is a projective object bound into a `ProjectiveObjectUserControl` and placed in the main area of the view (see *Subsection 5.8.3*).
5. `ProjectiveObjectUserControl` provides projections in which a user can edit the file.

The flow points out that everything revolves around a projective object that is stored internally in the DSLPed editor. An input file is deserialized into this object and an object is serialized back when it is required. An editor does not depend on the environment where it is used since it interacts with it via a well-defined interface.

However, in a flow, we omitted the most fundamental step. Since the editor is kept universal and is not aware of any domain-specific language, how does it

know the data type of an object which the file should be deserialized to. The type is heavily dependent on a language of a source file. This is a subject of the *Section 5.9*, where we will deal with the injection of this missing information into the DSLPed editor.

5.9 DSLPed Editor Extensibility

We said that DSLPed editor is a universal component that does not know any domain-specific language nor its projections. This information must be injected to the editor in order to let it recognize the given languages. This information is called DSLPed Add-in and before we describe how the injection works, we will summarize our expectations of an add-in with respect to our current DSLPed design.

5.9.1 DSLPed Add-ins

A projective component presents a projective object under one of its projections. Within a projection, there can be another projective components aimed at projections of objects lower in the AST hierarchy. These projective components are created automatically in a time when we ask for a projections of a given projective object, therefore, we do not have to care about their creation.

If we used a `ProjectiveObjectUserControl` as a main graphical control of an editor, the only challenge is how to get the root projective object, which we need to bind to that `ProjectiveObjectUserControl`. We mainly need a projective type to which we can deserialize the incoming XML file. Thus the information that must be injected to the editor must carry all available projective types and a logic to decide which one to choose for the opening of a specific XML file. A DSLPed add-in is a component that wraps this information.

The *Figure 5.16* demonstrates a workflow inside the DSLPed editor. When the XML is opening, the editor finds a proper projective type within all available add-ins, grabs an appropriate one and deserializes XML content into a projective object. That object is afterwards bound into the `ProjectiveObjectUserControl` that constitutes a main control of the DSLPed editor.

The result of our summary is that in the DSLPed add-in, there must be stored a projective type. However, the selection logic of an add-in and corresponding projective type to choose for a specific XML file requires additional parameters. Let us discuss that in the following section.

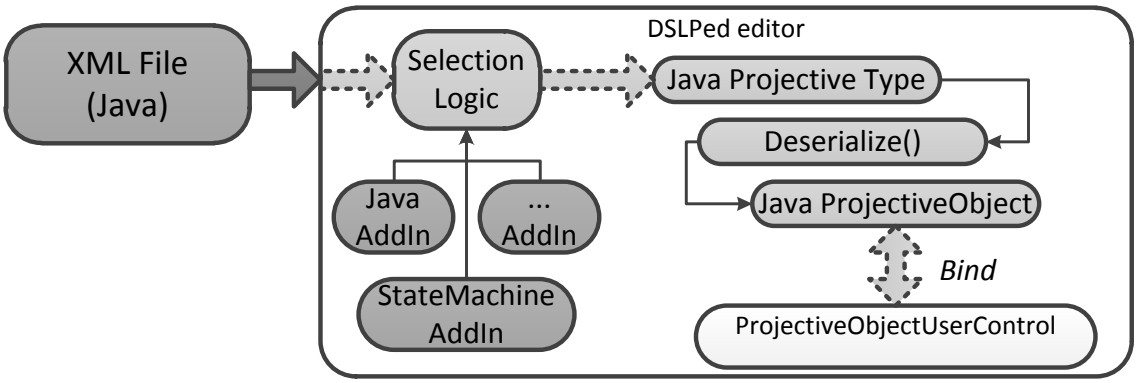


Figure 5.16: Workflow of an instantiation of the DSLPed editor using add-ins.

5.9.2 Add-in Recognition

The add-in and a corresponding projective type must be recognized by a selection logic based on the name of the opening file, or the content of this file. Such a logic is described in this section.

In general we have two pair the projective type with some unique identifier of a file. Basically we have two options, either pair it with a file extension or a namespace of its XML content.

5.9.2.1 Pairing with Extension

A file name is provided with a file extension. This might be a hot candidate for the selection logic, simply pair the file extension with a projective type is definitely a tempting solution.

However, later in this thesis, we will show that the DSLPed designer is installed in MSVS in a form of MSVS package. A package must be registered and paired with a specific file extension. Microsoft Visual Studio is thereby notified that we require to open a file of a specific extension in the given designer and maintains a convenient Windows registry key. The problem is that in our case a file extension is not registered statically together with the DSLPed designer, but it is injected to the DSLPed designer by MEF from another MSVS extension (from MSVS add-in). We faced up serious problems when we attempted to register and pair already installed package with a new file extension. Due to a lack of MSVS core documentation, we can just guess that it requires the changes of an appropriate key within the Windows registry. Since MSVS probably does not provide a public interface for such a change, these would have to be done manually.

After this survey, we decided to reject the pairing of projective type with the file extension till the time a MSVS documentation will clearly describe how to register new file extensions to already installed packages.

5.9.2.2 Paring with Namespace

A XML file of a specific DSL can be provided with a default XML namespace (`xmlns` attribute). We can pair this attribute with a projective type. To achieve this pairing, we have to store the `xmlns` attribute within the DSLPed add-in.

The selection logic simply finds the `xmlns` attribute of a file to be opened and compares it with corresponding attributes in all available add-ins. If it succeeds, it returns the projective type stored in the add-in. The *Figure 5.17* demonstrates the flow of a selection logic.

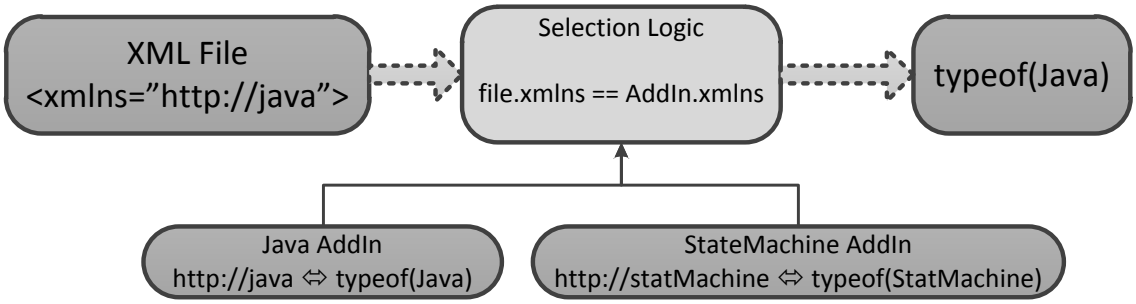


Figure 5.17: Selection Logic: If the *xmlns* of a file on the input matches defined namespace of an AddIn, a corresponding type is used for the deserialization of the file. On input there is a http://java *xmlns*, thus a Java type is used for the deserialization.

5.9.2.3 Namespace Wins

In this section, we analyzed that a DSLPed add-in must contain a unique identifier. This is used by a selection logic for finding a projective type to deserialize a given file. We concluded that a unique identifier will represent a namespace of the XML file content since the recognition by the file extension is technically impossible under the current DSLPed design.

5.9.3 Add-In Structure

We have analyzed the requirements of DSLPed add-ins in the last two section. Based on them, we will describe the structure of a DSLPed add-in.

In DSLPed, each add-in is a pair of a class and metadata object. Class implements an empty interface `IDslPedAddIn` and metadata object is of a type `IDslPedAddInMetadata`.

In fact all necessary information is stored in the metadata that explains the capabilities of a class implementing the `IDslPedAddIn`. Metadata is an object implementing a metadata interface. They can be seen as a list of attributes that describe

a specific class and are accessible even without instantiating that class. The attribute names match the names of properties within the specific metadata interface. The metadata interface is defined as follows (Only the crucial parts of interface are presented):

```
public interface IDslPedAddInMetadata {
    /// <summary>Projective type which represents the language</summary>
    Type ProjectiveType { get; }

    /// <summary>Uniqe XML namespace</summary>
    string Namespace { get; }
    ...
}
```

Listing 5.15: A sample of *Beeps* model

- ***ProjectiveType*** property is projective type of the root language construct.
- ***Namespace*** is a unique XML namespace paired with the ProjectiveType.

Beside that parameters, there are several others. Like a *Name* that refers to the name of an add-in, or *Description* that briefly describes what is the add-in about. These are more or less used for the integration purposes. DSLPed editor just ignores them. An application that provides add-ins and injects them to DSLPed editor can for instance visualize all available ones by name and their description.

5.9.4 Add-in injection

In this section, we will describe how this injection works.

The injection is happening in the constructor of a `DSLPedEditor` class via the collection of add-ins organized in a type `Lazy<IDslPedAddIn, IDslPedAddInMetadata>`. Let us scrutinize the add-ins data type first.

- `Lazy` is a .NET object type that supports a laziness - lazy instantiation. A creation of an object is postponed till the time it is actually accessed.
- `IDslPedAddIn` refers to an empty add-in interface.
- `IDslPedAddInMetadata` refers to metadata of the add-in class.

Object of a type `Lazy<IDslPedAddIn, IDslPedAddInMetadata>` represents a class implementing an interface `IDslPedAddIn`, which is attributed by a metadata `IDslPedAddInMetadata`.

Two properties of a `Lazy` type worth our attention. `Metadata` property returns the metadata of a class implementing an `IDslPedAddIn` and a property `Value` instantiates this class. Thus via this object, we can access some information about a DSLPed add-in without instantiating it. Its instance is created when we firstly call the getter of a property `Value`.

5.9.4.1 Benefits of Add-in Type

One may argue that the proposed data type is just too complicated and we can return the required metadata object by a convenient method in `IDslPedAddIn`.

Basically yes. We can instantiate all add-in classes before, and put them to the constructor of a DSLPed editor. After all, an interface `IDslPedAddIn` is empty, thus the instantiation of an add-in class should not be probably too expensive.

On the other hand, the add-ins give the editor an information about a way how it can handle a specific domain-specific language. When we want to maintain a code of a DSL, an editor must recognize the language and chose the suitable add-in to instantiate. The rest of add-ins remains unused and therefore it does not make a sense to keep them in the editor for further use.

5.10 Integration to Visual Studio

In the *Section 4.9* we analyzed how a projectional editor can be integrated into MSVS and thereby satisfy a decision step D8. The last section introduced the concept of DSLPed add-ins as units that hold the definitions of new languages. We only did not describe how these will be distributed and consumed by installed instance of a DSLPed editor that we created in the *Section 5.8*. Thus we will deal with that in the current section.

Furthermore, we analyzed that MSVS projectional editor must be integrated with another MSVS components. The way how DSLPed cooperates with another MSVS parts is an objective of this section as well.

5.10.1 DSLPed Add-ins Deployment

`DSLPedEditor` (see *Section 5.8*) is a universal component that does not contain any projections nor DSL definitions yet it recognizes various languages and even presents them in their custom projections. A suitable way to achieve such a behavior is to inject the missing information into an editor. We concluded that this information can be distributed in a form of DSLPed add-in, whereas a constructor of `DSLPedEditor` expects the collection of DSLPed add-ins in its parameter. Thus we moved a problem onto another layer.

The one who creates `DSLPedEditor` must instantiate it with a collection of DSLPed add-ins. Since this is happening in the wrapper of a `DSLPedEditor` and a wrapper belongs to an application that hosts this control, the problem of gathering DSLPed add-ins must be solved on the level of a hosting application.

Desktop editor and MSVS designer serve as hosting applications. The available DSL definitions and projections are gathered in the bootstrapper process using MEF

and injected to the editor. In this case, when the hosting application decides to load the required information via MEF, it must handle this process on its own since the loading strategy may differ application to application.

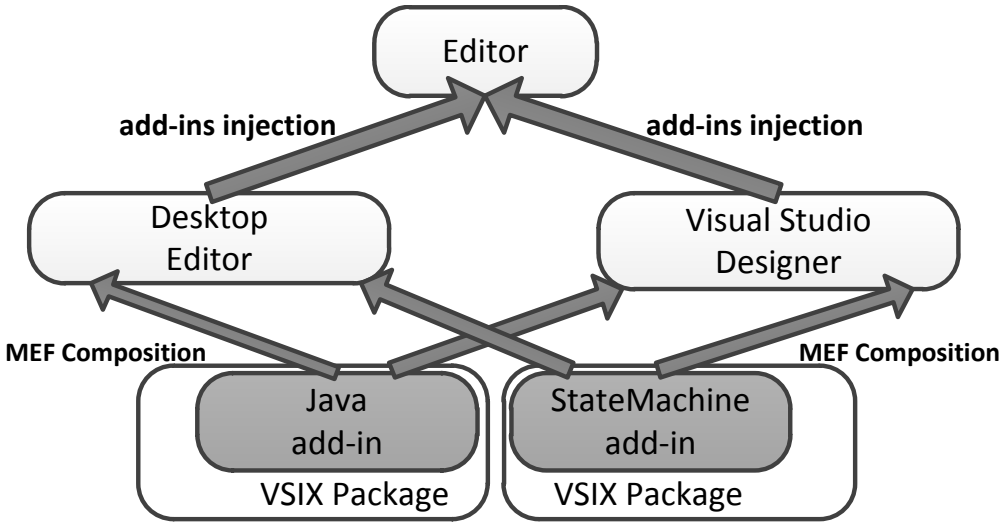


Figure 5.18: Integration of DSLPed add-ins via MEF

For instance, DSLPed designer is an extension to the Visual Studio that provides a MEF extension point. The developer releases new library where there is a VSIX package containing DSLPed add-in properly exported via MEF. When the Visual Studio is starting up, it loads all the MEF contributors that have been exported across all VS extensions. When the DSLPed designer is loading, its MEF extension point is filled with all the MEF exports that fit its configuration and contain DSLPed add-in. Before the DSLPed designer is initialized, it instantiates its main component `DSLPedEditor` with all available DSLPed add-ins presented in its extension point and thus injects the required knowledge of DSL and their projections in it.

The same information may be contributed to the desktop editor as well, however the MEF configuration will differ. Each assembly that contains a VSIX package exports a DSLPed add-in via MEF regardless it is loaded to MSVS or not. So when the desktop application is bootstrapping, it invokes the convenient MEF composition, loads the DSLPed add-ins and instantiates its main component `DSLPedEditor`.

An overall workflow looks like on the *Figure 5.18*. DSLPed add-ins are exported by MEF from the MEF components registered in a specific VSIX. The DSLPed add-ins can contribute to the DSLPed designer as well as to the DSLPed desktop. These construct a universal `DSLPedEditor` component and fill it with loaded DSLPed add-ins. Afterwards a whole designer or desktop editor starts to support the DSL defined in DSLPed add-ins.

The purpose of this section was to show, how DSLPed add-ins can be deployed

to a specific hosting application (like DSLPed designer or DSLPed desktop editor). We discussed that a proper VSIX package must be created and our DSLPed add-in must be registered in it properly. Finally, a VSIX package must be placed into a .NET library that creates a required VSIX package after it is compiled. The created package can be afterwards uploaded to Visual Studio Gallery from where it can be downloaded and installed.

5.10.2 DSLPed Add-ins MEF export

We already outlined that DSLPed add-ins will be registered in a certain VSIX package and exported via MEF. In this section, we will describe this mechanism in detail and provide corresponding code snippets.

In the *Section 5.9*, we introduced DSLPed add-in as a pair of metadata and a class implementing `IDSLPedAddIn` interface. It is exported via MEF as follows (see *Listing 5.16*):

```
[Export(typeof(IDslPedAddIn))]
[ExportMetadata("Extension", "beep")]
[ExportMetadata("Name", "Beeps")]
[ExportMetadata("ProjectiveType", typeof(Beeps))]
[ExportMetadata("Namespace", "http://beeps")]
[ExportMetadata("ResourceDictionary", typeof(BeepsResourceDictionary))]
[ExportMetadata("ImageUri", "Resources/Images/beeps.png")]
public class BeepsAddIn : IDslPedAddIn
{
}
```

Listing 5.16: DSLPed add-in MEF export

Each `ExportMetadata` attribute corresponds with a property member placed in the interface `IDSLPedAddInMetadata`. Some of them are mandatory, some of them not.

- ***Extension*** is not supported by DSLPed at the moment due to the arguments discussed in the *Subsubsection 5.9.2.1* .
- ***Name*** is a name of DSL included in add-in.
- ***ProjectiveType*** is a projective type which the file can be deserialized to.
- ***Namespace*** refers to the XML namespace which the projective type is paired with (as described in the *Subsubsection 5.9.2.1*).
- ***ResourceDictionary*** is an optional attribute that refers to a WPF Resource-Dictionary that can be used by an editor to style the projections with respect to defined templates.
- ***ImageUri*** is an relative path to an image representing an add-in.

The best practice of the DSLPed add-in development is to put this export into a file called *[Name of a language]AddIn.cs*. A project template released with

DSLPed creates this file automatically.

5.10.3 DSLPed Add-in Import

We said that a DSLPed add-in is wrapped in a `Lazy` object. One of the reasons to use `Lazy` type for DSLPed add-in representation is in fact its native support by MEF. To do the composition with all available DSLPed add-ins properly, we have to do the following import (see *Listing 5.17*). It is basically the same what is happening on the level of DSLPed designer or DSLPed desktop editor to gather all available DSLPed add-ins.

```
[ImportMany(typeof(IDslPedAddIn))]
public IEnumerable<Lazy<IDslPedAddIn, IDslPedAddInMetadata>> availableAddIns { get; set; }
```

Listing 5.17: DSLPed add-in MEF import

This import must be accompanied with a proper MEF composition which differ application to application, therefore, we will not pay much attention to it. The process looks similar to the *Listing 5.18* and its details can be found in any MEF documentation.

```
catalog = new AggregateCatalog();
catalog.Catalogs.Add(new DirectoryCatalog(addInsDirectory));
Container = new CompositionContainer(catalog);
Container.ComposeParts(this);
```

Listing 5.18: DSLPed add-in MEF import

5.10.4 Registration of DSLPed Add-in in VSIX

To wrap DSLPed add-in by a unit that is recognized by MSVS, a VSIX package must be created. When we use a wizard for the creation of a DSLPed add-in, it automatically creates a project of required type and preconfigures a VSIX manifest called *source.extension.vsixmanifest*.

Regardless the common attributes of VSIX manifest, which are set in accordance to the language, its content must be provided with a MEF component. An assembly which the VSIX manifest belongs to defines this component like on the *Listing 5.19*. This XML content is in fact created behind the scenes, thus we will not get in touch with it directly. We will most probably edit the VSIX manifest via a default MSVS editor.

```
<Content>
  <MefComponent>|\%CurrentProject\%|</MefComponent>
</Content>
```

Listing 5.19: Export of DSLPed add-in via MEF component

After the project implementing DSLPed add-in and containing VSIX manifest gets compiled, a corresponding VSIX package is created. This can be installed into MSVS on a common basis.

5.10.5 Error List Integration

Common error list provided in MSVS is integrated with an editor as well. In a desktop editor, it was implemented for testing purposes. An editor notifies its hosting application via MVVMLight message that something goes wrong in a projection. This notification is usually released from the method that validates a projection, thus on the level of its definition.

A hosting application (Desktop editor of MSVS designer) listens to this messages and properly notifies the corresponding error list to update an information in it. We have to point out that the communication using messages is happening via a channel called `Messenger`. Since all the opened editors in MSVS communicate over the same `Messenger` instance, their handling is kind of complicated and is not fully debugged. However the error notifications can be distributed this way.

5.10.6 DSLPed Add-in Project Template

Several predefined project templates are installed by default in MSVS. To extend DSLPed Designer or Desktop Editor with a new domain-specific language and its projections, we have to implement a DSLPed add-in. DSLPed add-in is nothing but a .NET library. To speed up its development and provide a developer with its recommended structure, a project template of DSLPed can be created.

DSLPed *IDE Integration*, part of DSLPed solution (*Subsection 5.2.5*) provides the implementation of such a template. It can be found in the *IDEIntegration/Templates/DSLPedProjectTemplates* project. Template is represented by a separated directory that contains a crucial file of a *.vstemplate extension. It describes the template according to the MSVS requirements [22].

The template is integrated to MSVS from the project *IDEIntegration/DSLPed-Vs2010Integration*, where it is attached as a project template content in the core file called *source.extension.vsixmanifest*.

Since the project template development is fairly described on [22], we will not discuss it in detail any further.

5.10.7 DSLPed Add-in Files Template and Wizard

When the DSLPed add-in is created using a project template and deployed to a targeted MSVS installation, the end-user of MSVS can use it (see *Section 5.1*). MSVS

will automatically open a file supported by an add-in in DSLPed Designer. For instance, a file containing the source code of a *Beep* language (see *Subsection 5.3.2*) will be automatically opened in DSLPed Designer if a corresponding DSLPed Beep add-in is installed in MSVS.

However, the end-user is not interested in the opening files only. He must be able to create new file containing the source-code of a required domain-specific language. Therefore, we need to provide a convenient way to create new files of languages that are supported by any DSLPed add-in installed in MSVS.

Similar issue is in MSVS usually solved by item templates, the user can simply select an item, which he is interested in and create it. To integrate such a template into MSVS, we usually have to deliver a prefabricated file representing the item. A file contains various placeholders that are replaced by attributes gathered in a time of item creation. A *.vstemplate extension accompanies this prefabricate to integrate template into MSVS.

In our case, each add-in would need to implement a DSL-specific item template, which would require to delegate the creation of whole item template to the developer of a DSLPed add-in. This approach would force the developer to be familiar with a MSVS extensions development and would significantly increase the risk of errors done in DSLPed add-in.

Thus in DSLPed we implemented a general item template that creates new DSL files using a wizard.

5.11 Implementation Conclusion

This chapter implemented a projectional editor analyzed in the *Chapter 4* and thus proved its proposal.

Language is described via XSD, out of which a class representing a model is generated using *Xsd2Code*. Projections are WPF controls properly bound into the model. Their definitions are kept in a separated partial class that naturally extends the model. Since each extended model belongs to one DSL, it is stored in an independent library and exported via MEF. The library itself is injected into MSVS in a form of VSIX. When a source code written in a specific DSL is opening, MSVS imports a suitable MEF component, instantiates it and presents in one of its projections.

Due to the limited size of this thesis, not all the technical details were discussed in deep, these are properly commented in the source code attached to this thesis.

Chapter 6

DSLPed Tutorial

The goal of this chapter is to go through the whole process of a custom DSL creation. We will pass the installation of DSLPed, creation and deployment of DSLPed add-in and finally, we will show how to use the developed add-in in MSVS. We will implement DSLPed *Beeps* add-in, it will allow us to maintain sources written in *Beeps* language that was introduced in the *Subsection 5.3.2*. Even though the language is simple, the detailed description of its add-in is out of the scope of this thesis. Therefore just the core issues will be discussed while the entire sources of the add-in can be found on an attached CD (see *Appendix 9.2.1*).

6.1 DSLPed Installation

To develop new add-in, DSLPed must be installed before. DSLPed is provided in a form of VSIX file that must be handled on a common basis.

1. Let us copy the file *DSLPedVs2010Integration.vsix* from the attached CD
2. After double-click it, we follow the installation instructions
3. To verify the installation of VSIX was successful, we can open MSVS *Extension Manager* (*Tools/Extension Manager*)
4. A corresponding entry DSLPed (DSLPed integration for Microsoft Visual Studio 2010) should be listed among the extensions

6.2 DSLPed Desktop

The development of a DSLPed add-in requires quite a frequent manual testing. To speed this process up, a DSLPed desktop editor can be used. The application is wired up to look for the add-ins stored in the subfolder *AddIns*. All add-ins located in this folder are automatically injected into the desktop editor and are ready to be used in a similar way as in MSVS.

On the *Figure 6.1*, there is a snapshot of a DSLPed desktop editor. All available DSLPed add-ins are automatically loaded to the area marked by the border. Three add-ins are loaded: *beeps*, *java* and *peoples* add-in. A click on an appropriate Ribbon button leads directly to a creation of a default language instance and a corresponding source file.

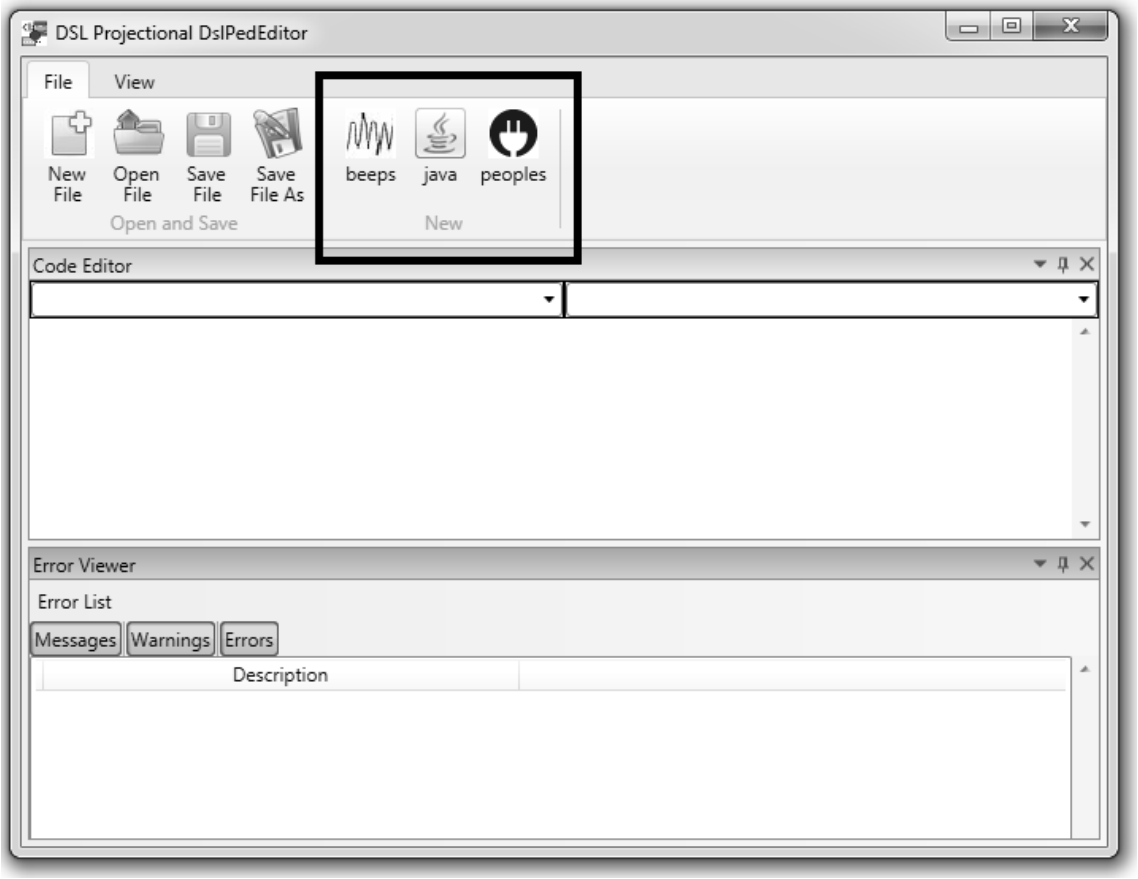


Figure 6.1: DSLPed Desktop Editor with marked language extensions

6.3 Beeps Add-in Creation

Once the DSLPed is installed in MSVS, the implementation of new DSLPed add-in can begin. A default template for the add-in creation is prepared in MSVS.

Creation of Dummy Add-in

1. Let us open common *New Project* dialog in MSVS (*File/New/Project*)
2. After selecting a template called *DSLPed AddIn Template*, a wizard similar to the *Figure 6.2* will be opened
3. We are prompted to specify valid name of our language (Beeps) and a suitable XML namespace

4. After filling the necessary attributes and confirming the wizard, a preconfigured MSVS project is created

Figure 6.2: DSLPed New Add-in Wizard

5. As it can be seen on the *Figure 6.3*, a MSVS project contains various specialized files.

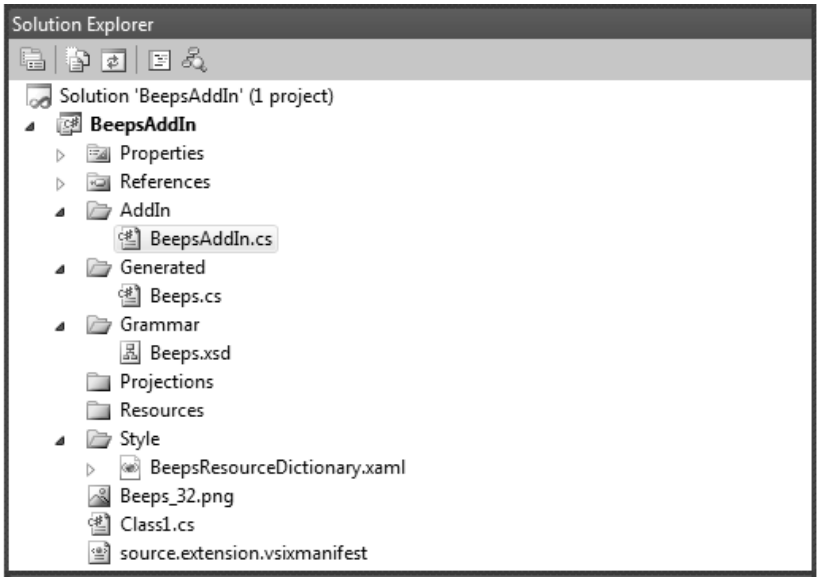


Figure 6.3: DSLPed AddIn

- ***BeepsAddIn.cs*** file refers to a class that was discussed in *Subsection 5.10.2*. It exports several MEF attributes recognized by DSLPed desktop and DSLPed designer.
- ***Beeps.cs*** is a dummy implementation of *Beeps* model. We have to replace it with a model produced by *Xsd2Code* later on.
- ***Beeps.xsd*** represents a dummy grammar of our language. We have to reimplement it appropriately.

- ***BeepsResourceDictionary.xaml*** is a place where we can put the styles and templates used by WPF projections.
- ***source.extension.vsixmanifest*** denotes to a file responsible for future deployment of our add-in and convenient VSIX file creation.

Missing References

When we attempt to compile the created project, tons of errors will appear. The reason is that an add-in is missing the references to DSLPed core structures. To fix it, we have to add the references to the following assemblies (all of them can be found on an attached CD):

- *DSLPedCommon.dll*
- *DSLPedControls.dll*
- *DSLPedFramework.dll*
- *GalaSoft.MvvmLight.WPF4.dll*
- *GalaSoft.MvvmLight.Extras.WPF4.dll*

The last two libraries are kind of important since a DSLPed depends on a toolkit *MVVMLight* that constitutes its core structures.

Xsd2Code Model Generation

Since an add-in is provided with the dummy grammar and a language model only, we have to create a convenient XSD describing our language. Afterwards a model can be generated using an *Xsd2Code* out of it. If *Xsd2Code* is missing in MSVS, it must be installed before.

1. Let us change the *Beeps.xsd* to establish a proper *Beeps* grammar
2. Right click the file and select an option *Run Xsd2Code generation*
3. A dialog window similar to the *Figure 6.4* will appear
4. The important parameters to be set are marked by borders
    - *EnableDataBinding* must be set true to enable binding of a language model to projections
    - *GenerateXmlAttributes* will generate Xml attributes needed for the deserialization and serialization process
    - *CollectionObjectType* can be set in general to any collection type, however `ObservableCollection` is recommended
5. After settings the generation process and clicking generate, the *Beeps* language model similar to *Listing 6.1* will be created

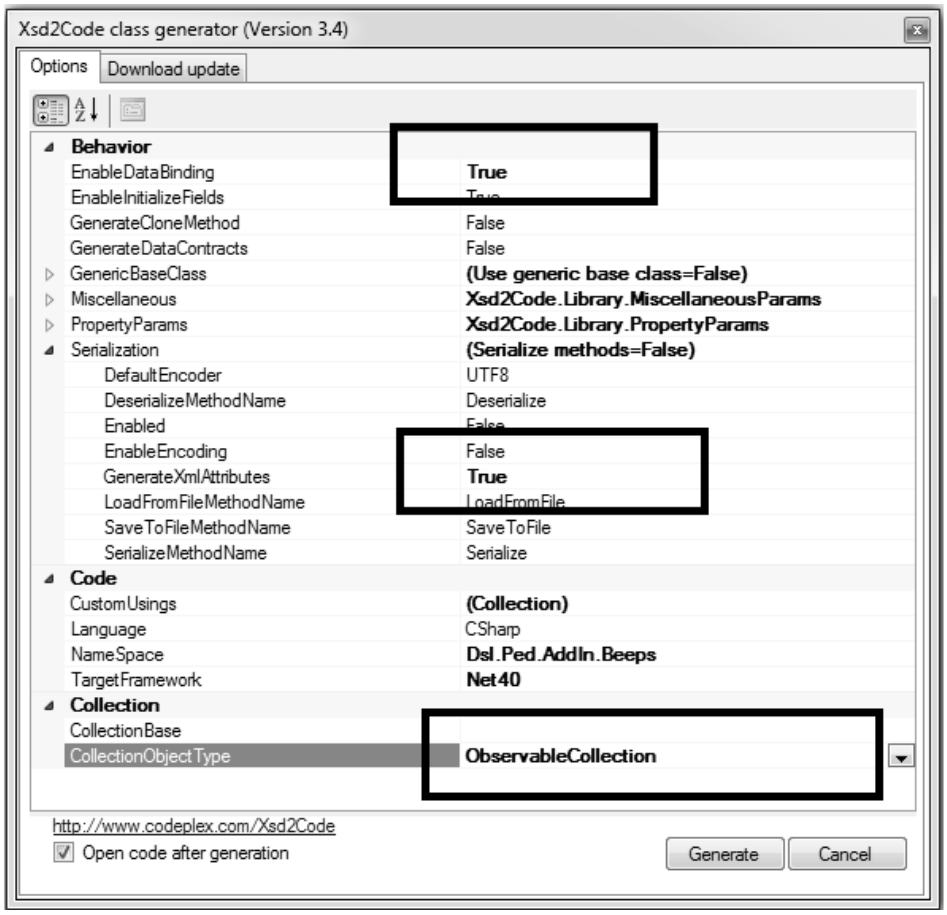


Figure 6.4: Xsd2Code generation

```
public partial class Beeps : INotifyPropertyChanged {
   public ObservableCollection<BeepsBeep> Beep {
     ...
   }
   ...
}

public partial class BeepsBeep { ... }
```

Listing 6.1: Generated *Beeps* model

Association of Model with Projections

Currently an add-in contains a XSD file representing the *Beeps* grammar and a corresponding C# model generated out of the XSD. To enhance a model of new projections, the scenario outlined in *Subsection 5.3.3* must be applied.

First we have to decide, whether a shared projection implemented in *DSLPed-Framework* library can be reused and instantiated via a certain factory method, or if we will create a brand new projection. In our example, we will create two different projections. One using a factory method and a custom one presenting data in a sim-

ple concatenated list. Since the concatenated list is more or less a common WPF control, we will not describe its implementation, it can be found in the attached source code.

For now, we will create new file called *Beeps* that implements a partial `Beeps` class (see the *Listing 6.2*).

```
public partial class Beeps : IProjective {
    public List<IProjectionContainer> GetProjectionContainers() {
        return new List<IProjectionContainer>() {
            new ProjectionContainer("Linked_List")
                .RegisterProjection("LinkedList", new LinkedListProjection(this, "Beep")),

            new ProjectionContainer("Plain_Text") { ForceRefresh = true }
                .RegisterProjection("PlainText",
                    TextProjectionFactory.GetSyntaxHighlighterProjection(
                        this,
                        ConvertToPlainText,
                        sh => {
                            sh.KeyWords = BeepsResources.GetKeyWords();
                            sh.KeyWordsDelimiter = BeepsResources.KeyWordsDelimiter;
                        }))
        };
    }
}
```

Listing 6.2: Definition of *Beeps* model projections

This is in fact all we have to do to associate a class `Beeps` with new projections. Similarly, a class `BeepsBeep` must be enhanced of projections. It this case a simple event-driven projections will be defined.

```
public partial class BeepsBeep : IProjective, IDataErrorInfo {
    public List<IProjectionContainer> GetProjectionContainers() {
       return new List<IProjectionContainer>() {
          new ProjectionContainer("Number")
              .RegisterProjection("Number",
                  ValueProjectionFactory.GetLabelProjection(
                      this,
                      () => frequence))
              .RegisterProjection("Combo",
                  ValueProjectionFactory.GetComboBoxProjection(
                      this,
                      () => frequence, GetFrequencies()))
              .RegisterTransition(Label.MouseMoveEvent, "Number", "Combo")
              .RegisterTransition(Control.MouseLeaveEvent, "Combo", "Number")
          };
    }
}
```

Listing 6.3: Event-driven projection definition

Add-in Deployment

Now the DSLPed add-in is mainly finished. We can compile it and deploy the VSIX file to the end-user installation of MSVS. The installation follows the same scenario as outlined in the *Section 6.1*.

6.4 DSLPed Add-in Use

As soon as the add-in is successfully installed to the target MSVS, we can simply create new source file of our *Beeps* language and edit it within one of the designed projections. The creation of desired file can be achieved as follows.

1. Right click the project in MSVS
2. Select *Add/New Item/ DSLPed File*, the following wizard will appear

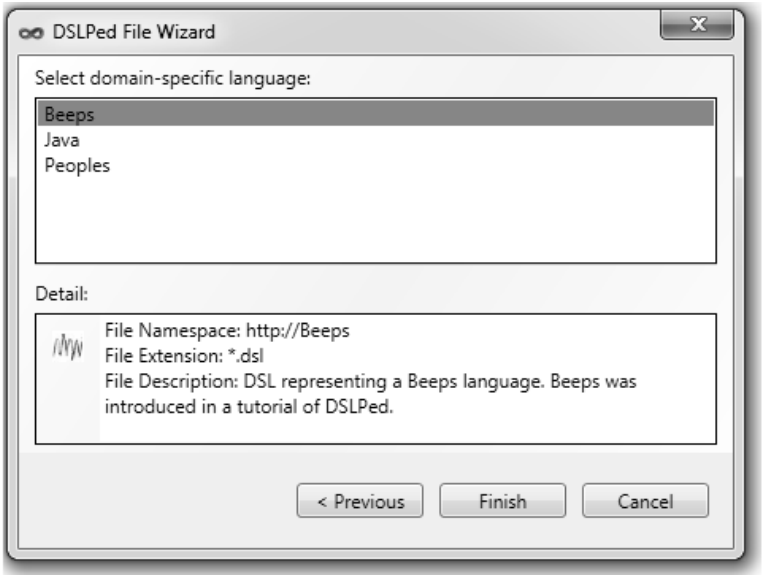


Figure 6.5: DSLPed new file wizard

3. After clicking *Finish*, new file of the required language will be created and an editor will be opened for it
4. When we hold a *Shift* key and click into the editor area. A projective control immediately under the mouse is selected and we can change its projections in the context menu or in the combo box located in the upper right corner of an editor.
5. Different projections of a *Beeps* language are displayed on the *Figure 6.6* and *Figure 6.7*.

The aim of this chapter was to provide a tutorial that outlined the process of DSLPed installation, its add-in creation and usage. A tutorial operates on a

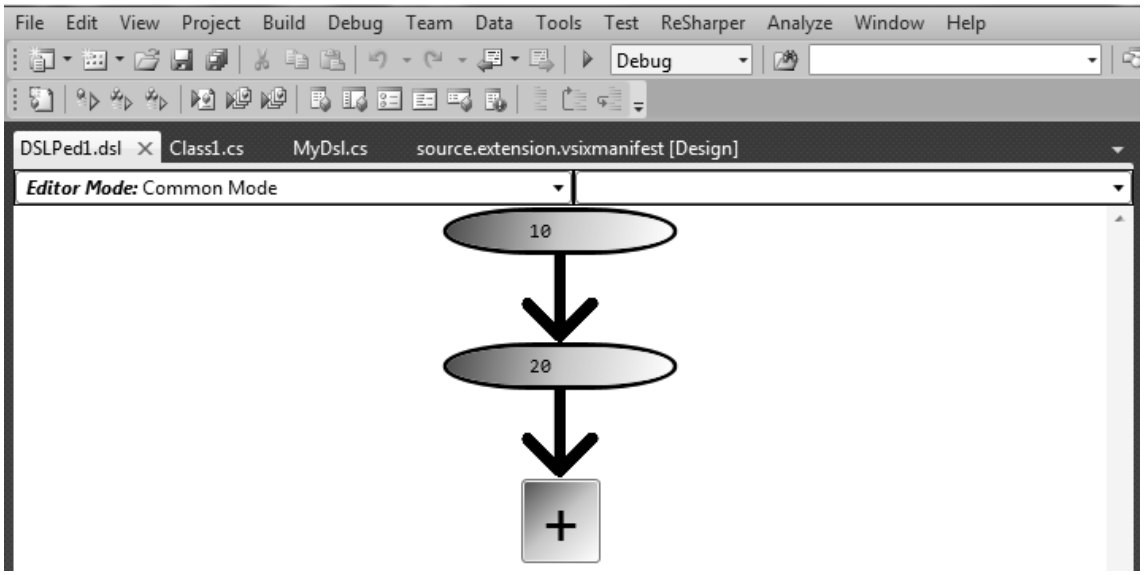


Figure 6.6: Beeps concatenated list projection in Microsoft Visual Studio

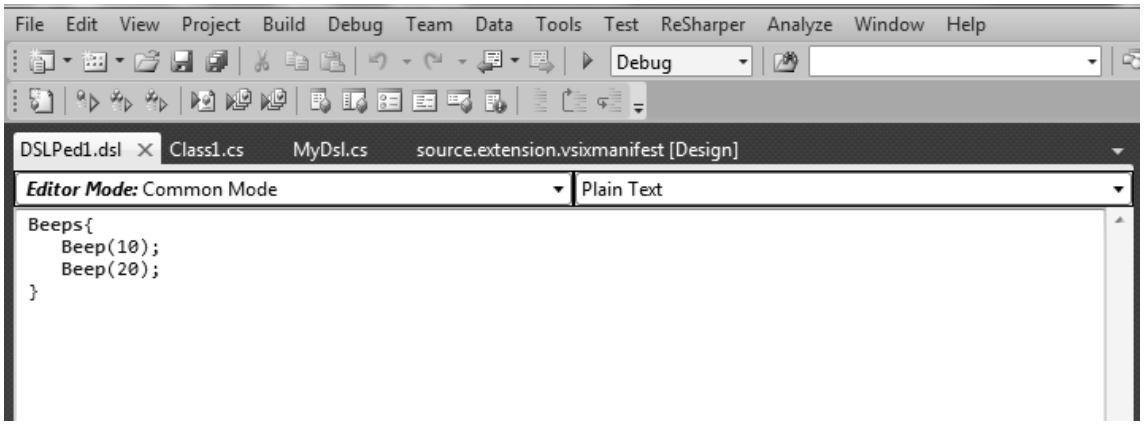


Figure 6.7: Beeps plain text projection in Microsoft Visual Studio

conceptual level, it can be hardly considered as a common tutorial for the step by step creation of a DSL extension.

Chapter 7

Related Work

The comprehensive proposal of a projectional editor was presented in this thesis. However, this is not the only existing projectional editor. The topic of projectional editing is new and thus not a large number of projectional editors and tools related to DSL handling exists. Most of them are just research projects in a state similar to DSLPed. These were confronted in a language workbench competition [23], where they were compared according to various criteria. Some of them worth mentioning.

MPS [15], a workbench released by JetBrains is a powerful Java-based tool for designing own DSL and suitable rich editors generation. A text-based projectional editor with a code-completion, semantics and type checking can be generated easily using MPS. Moreover, it allows to create generators to compile a custom DSL into various languages. MPS is aged and well-thought project mainly targeted for Java and C programmers. Despite the fact that it is not provided in a form of Eclipse plugin, it cannot be ruled out that it does sometimes happen. Although MPS is much more advanced project than DSLPed, it does not support C# and is not integrated in MSVS as DSLPed.

PrEdE, a projectional editor for Eclipse Modeling Framework[1], was proposed as a paper [14] during the Eclipse workshop few years ago. It is an editor that can represent the same data using multiple notations while Java is used for the projections creation. Unfortunately no official page of the project exists, therefore it is reasonable to assume the project is dead.

Microsoft has released *DSL Tools*, Domain-Specific Language Tools [21]. First it was shipped as an external package for MSVS 2005, it was joined to the MSVS 2008 SDK later on. It is a tool for a creation of custom DSL and suitable MSVS designers for them. The language definition is described using a kind of an Entity-Relationship diagram, which is called metamodel. MSVS designer is generated out of it. Nevertheless, the designers generated by DSL Tools are not projectional. It

[1] Modeling and code generation facility for a creation of tools that are provided with a structured data model

generates an editor that provides one projection only and one can hardly consider it as a projectional editor. While it supports editing of a custom language in a form of diagram, it cannot be simply switched to another visual form. The custom projections creation and their mapping to a specific language construct is also not that straightforward.

Chapter 8

Evaluation

The main objective of this thesis is to analyze approaches to a projectional editing and solve a problem of designing and implementing a projectional editor extendable with custom DSL. The deeper analysis of a problem was done in the *Chapter 4* while the prototype of a projectional editor was implemented and described in the *Chapter 5*. Thus from this point of view, the objective is satisfied. The only thing missing is to evaluate the crucial decisions of the thesis and briefly discuss, which of them were sensible and which has brought us far more work than we expected and if we were dealing with them again, we would probably apprise them differently. Moreover, we have to assess the thesis from its contribution point of view. How much effort would it take to create a projectional editor of a custom DSL from scratch, without the proposed DSLPed tool.

8.1 Design Suitability

In this section, we will concentrate on the crucial decisions done in this thesis. We will go through them step by step in the same order as we proceeded with the problem decomposition in the *Section 4.1*.

≫ *D1: Language described by XSD*

First we tackled a problem of describing the language. We analyzed that such a description can be expressed in many ways. Since a tool *Xsd2Code* for elegant generation of a class out of the XSD was introduced and since we concluded that a language model is in fact a class, we picked finally XSD as a most appropriate form for a language description. It has proved to be a good decision. The language model represented by XSD was automatically serializable, enabling us to facilitate the integration into MSVS and allowed us to load and store the source codes of a given DSL easily. On the other hand, XSD brings several limitations which do not

allow us to describe all languages. However, domain-specific languages are usually composed of simple language constructs and thus XSD is a suitable form to describe them.

≫ *D2: Projections defined in WPF*

WPF was chosen as the main technology for a definition of projections. A language construct is usually provided with several projections and these must be synchronized. Thanks to data binding, which was considered as a main benefit of WPF, no additional effort was needed to achieve the synchronization.

Originally we were struggling with a problem of mapping language constructs to the projections. Since we managed to represent a language by a class, we in fact transformed a problem to another one - mapping of class elements to projections. That was solved naturally by a native MVVM pattern, which WPF is based on.

For the above mentioned reasons, WPF was evaluated as a suitable technology for the definition of projections.

≫ *D3: Projective object associating the language with projections*

Projective object was introduced as an object representing a specific language construct that is aware of its own projections. By proposing projective object, we rejected a second approach of clear model separation from its projections. Under the current design projective object fits to our needs, on the other hand, if we wanted to persist a user settings of currently set projections, we would probably need to provide an additional data structure for it. This might be an easier task under the second contemplated approach - to keep projections separated from a model.

≫ *D4: Event-driven projections via routed events*

We decided to organize projections into independent units and establish an event-driven workflow among them. It has proved to be a nice concept, nevertheless its realization faced problems on a technical level. This mechanism can be achieved via routed events in WPF. Unfortunately, some of them are handled automatically by WPF and their subsequent catch can be kind of tricky. For this reason, the event-driven hops between projections are sometimes painful and will require a future customization.

≫ *D5: Projective component for managing projections*

Projective component was implemented as a `ProjectiveObjectControl` in DSLPed. We were on a wrong way for a while. At first, we linked the projective

object directly with so called `ProjectiveObjectControl`, but we realized the limitations of this decision soon. `ProjectiveObjectControl` was too tightly coupled with the projective object, thus it was predetermined for the use by the projective object only and we lost its universality.

Therefore, we came with an independent control called `ProjectiveControl` which did not know anything about the projective object to treat it in a special way. New possibilities have opened to us. We were able to create projections within any XAML code, which in general had nothing to do with projectional editors. In this context, `ProjectiveControl` was used as a switch between various XAML elements.

Nevertheless, we did not avoid to use the `ProjectiveObjectControl` since we needed to bind it with projections of a projective object. Thus we reimplemented it and made it a wrapper of `ProjectiveControl`, which fed the nested control automatically with the bound projections.

This transformation to the universality was a crucial decision since the projectional editors were given another dimension, they have become useful in another fields of work than just in a context of projectional editors.

≫ *D6: Projectional editor as an independent component*

Just from the requirements on the thesis it was evident that it will be necessary to test the behavior of developed editor and projections manually and quite often. Thus it was reasonable to optimize a time for the startup of an editor to proceed with manual tests. If we were developing the editor within the MSVS only, we would need to wait till the whole testing environment of Visual Studio starts up. Since this was a non-sense time-consuming operation, we decided to host an editor within a desktop application, whose startup was a way faster than the startup of a whole testing IDE. To achieve the similar behavior in the desktop application as well as in the IDE, we implemented projectional editor in a form of an independent component that can be wrapped by any hosting application.

At least from the testability point of view, projectional editor in a form of an independent WPF component was concluded as a good decision step.

≫ *D7: Extensibility with the languages and projections via MEF*

MEF was chosen as a crucial technology to enable extensibility of the editor. We found it as a handy, lightweight, and a well-documented tool which has perfectly fitted to our solution.

≫ *D8: Integration to MSVS via VSIX*

To successfully integrate proposed projectional editor with MSVS, we needed to customize various MSVS components. If MSVS was an open source project, new

components could be integrated into it much more easily. Its official documentation is not sufficiently detailed to balance a time spent with the hopeless attempts to integrate new components into it. On the other way around, the MSVS 2010 is much more configurable and extendable than its predecessors and hopefully it will continue to do so in its later versions as well. Thus, the integration of DSLPed into MSVS was sometimes a painful job, which has required quite a deep investigation of seemingly unrelated IDE parts.

≫ ***D9: Analysis of projections and their sharing across languages***

Finally, we analyzed the projections themselves and concluded that it is likely to be useful to design general projections that can be used across different languages. This approach might be useful when several developers contribute to a shared library of projections. If these are sufficiently tested, such a reuse of projections can significantly speed up creation of projectional editors of languages with the similar characteristics.

8.2 DSLPed Eligibility

DSLPed is a tool for design and creation of projectional editors. We would like to evaluate how the creation of a projectional editor is facilitated with its use. It is quite a hard task to provide an exact metric for such an evaluation, if there exists any reasonable at all.

Therefore, we decided to discuss how much effort would it take to implement a projectional editor of a given DSL on its own, without the use of a DSLPed tool. We will try to emphasize the aspects in which the use of DSLPed accelerates or eases the development of a projectional editor under the same initial conditions.

Same Initial Conditions

Regardless the time spent on the investigation of technologies and possible approaches to solve the problem in this thesis, let us declare a language definition is provided in a form of XML, C# is a language used for the implementation and WPF is a technology for the creation of required projections. Moreover, the language model in a form of C# class has already been successfully generated out of the XML using a *Xsd2Code*-like tool, which we applied in DSLPed solution and described in *Subsection 5.3.1.* Thus it thereby gives us the initial conditions under which we will implement a projectional editor form the scratch, without DSLPed.

Projectional Editor without DSLPed

At the very beginning, we have to use WPF to create a view part of already created model. Let us call it the projection. Since we are interested in projectional editors, we will create an additional view parts to have an alternative projections of our language model. Just one projection can be seen in a period of time, therefore in our XAML code, we will design a kind of a visual switch to maintain the visibility of given projections. A projection is a visual representation of the language constructs, which are composed of another language constructs requiring to be displayed in certain projections as well. Thus we will probably copy the visual switch into the XAML code of projections to maintain their subprojections switching as well. It is obvious that the same problem will appear on several levels of projections and subprojections and on each level, it will be handled by copying the visual switch. Since the visual switch does still the same job, it would be reasonable to implement it as a universal WPF component. This is exactly what DSLPed provides - the `ProjectiveControl` to handle such a switching among projections. Without DSLPed, we would have to design and implement it on our own and furthermore, we would have to establish a mechanism to automatically reflect any changes done within one projection to others.

In the context of a projection, there are usually routed events fired. Sometimes, we will be dealing with a necessity of transition to another projection in dependence on these events. This mechanism should be also implemented in a component maintaining the projections. It is evident that such a component becomes much more complicated than it seemed to be. After all, it is a major subject of the whole *Chapter 4* and *Chapter 5*. Therefore, its creation is not as simple task as would expect.

Projections represent the language constructs in a form of WPF Controls. Even though most of the languages and their constructs require projections tailored to their specific needs, it is reasonable to have a pool of those projections that can be recycled and serve as a general-purpose projections. It is in fact a set of elementary projections, which the complex ones are composed of. If we omitted its creation, we would most probably end up in a code redundancy originated by creating the similar basic projections frequently. Therefore, it is reasonable to implement this collection as a set of basic building blocks. A collection of various elementary projections and their factories is provided within DSLPed, and without it, we would need to design and implement it from a scratch.

As soon as we deal with the creation of projections, their binding to language model and implementation of a component managing the projections, we will be facing another problem - integration into MSVS. DSLPed contains not just an implementation of the MSVS projectional editor, it provides various project and item templates for the handling of given language files. All these parts would need

to be created from scratch and since they expect quite a deep knowledge of the MSVS core, it might be a hard, time-consuming job. DSLPed is trying to shield the developer from the complexity of MSVS by letting him implement only extensions to the existing projectional editor. Thus it does not assume developer's deeper knowledge.

Eligibility Conclusion

In this thesis, we proposed a solution to facilitate and speed up a creation of a projectional editor of a given DSL. Its prototype DSLPed has been implemented and described properly. In this section, we focused to the aspects of a proposed solution, which might be considered as crucial in the development of projectional editors. We pointed out that the lack of several fundamental components would not impede creation of projectional editor, nevertheless its development would be much more painful and time consuming than with the use of DSLPed.

Chapter 9

Conclusion and Future Work

The main goal of this thesis was to analyze potential approaches to a projectional editing and implement a convenient projectional editor. Based on the analysis in *Chapter 4*, we concluded that a projectional editor can be designed on a more universal level. Thus we implemented various components useful in distinct WPF applications regardless their relation to the projectional editors. DSLPed, an editor implemented in this thesis, is then just a specialized application built from those universal components.

Proposed design enabled an easy integration of an editor into MSVS as well as into a standalone application. Thus both the integration accompany this work, standalone editor for the testing purposes and MSVS designer to fulfill an overall goal of this thesis.

9.1 Prototype Implementation

This thesis is supplemented by a prototype C# implementation of a projectional editor for MSVS. The implementation was released as a GNU project and its actual version can be found at http://dslped.codeplex.com.

9.2 Future Work

The main future task is to enrich the proposed solution of new sophisticated projections and ease their configuration and instantiation. DSLPed projectional editor should be enhanced with new features, especially to manage multiple projections at once and persisting their current settings.

Beside that, a process of DSLPed add-ins creation should be improved. This can be achieved by designing a self-editor briefly described in the next section.

9.2.1 Self-editor

Under the current design, the projections are defined in a C# code, which is written manually. The code that defines projections exhibits the same characteristics regardless of what projections it describes. It always returns projections via a given method and usually instantiates them using a common factory method. On closer inspection, we can find that this is a convenient task for a certain domain-specific language. An overall goal of such a language is just to describe projections of a language constructs and shield the programmer from the need of writing the C# code on his own. A required C# code would be simply generated out of that DSL code.

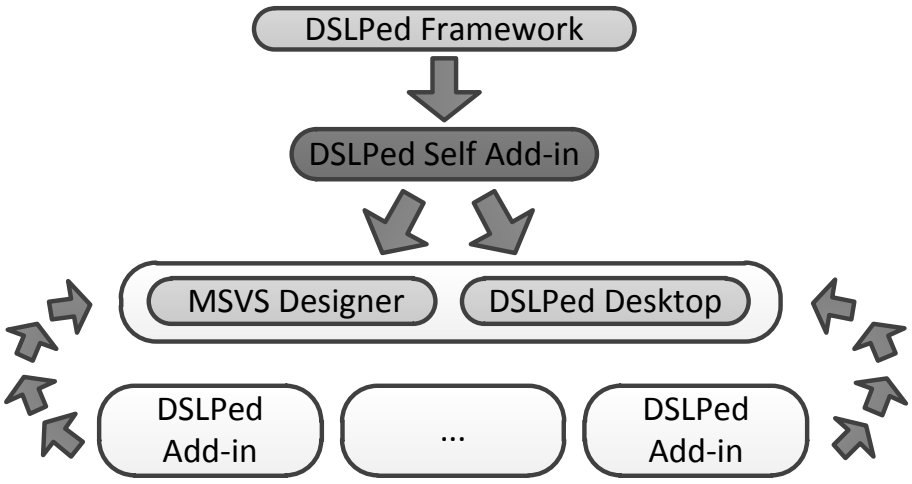


Figure 9.1: Outline of a self-editor

Thus an ambitious future task is to propose a grammar of such a DSL and provide a set of projections to handle its code. Within a DSLPed implementation, we have already experimented with T4[1] for the generation of C# code out of a language model. Therefore, the only task is to extend this implementation with respect to the DSL that will be proposed. The projections can be kept inside a DSLPed add-in in the completely same way like in another DSLPed add-ins. An add-in will be afterwards released together with a DSLPed package, injected to the MSVS and recognized by a DSLPed designer.

This idea of creating a DSL for describing projections of another DSL in fact results in a self add-in that is created using a DSLPed framework. It can be injected into a DSLPed designer (MSVS add-in) as well as into a DSLPed desktop editor. Within the designer or within the editor we can create new add-ins that can be injected into the same instance of DSLPed designer or a desktop editor. The *Figure 9.1* outlines that idea.

[1] Microsoft's Text Template Transformation Toolkit

Bibliography

[1] Fowler, M., *Domain-Specific Languages*. 1. edition. Addison Wesley, 2010. ISBN13 978-0321712943.

[2] Voelter, M., Solomatov, K., *Language modularization and composition with projectional language workbenches illustrated with MPS*. http://voelter.de/data/pub/VoelterSolomatov_SLE2010_LanguageModularizationAndCompositionLWBs.pdf, 2012.

[3] Voelter, M., *Embedded Software Development with Projectional Language Workbenches*. http://www.voelter.de/papers/Voelter-EmbeddedSystemsDevelopmentWithProjectionalLanguageWorkbenches.pdf, 2010.

[4] Jones, J., *Abstract Syntax Tree Implementation Idioms*. http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf, Department of Computer Science, University of Alabama, 2003.

[5] Gamma, E., Helm, R.,Johnson, R.,Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. ISBN 0-201-63361-2.

[6] Randolph, N., Gardner, D., Anderson, C., Minutillo, M., *Professional Visual Studio 2010*. 1. edition. Wrox, 2010. ISBN13 978-0470548653.

[7] Nayyeri, K., *Professional Visual Studio Extensibility*. Wrox, 2008. ISBN13 978-0470230848.

[8] Seemann, M., *Dependency Injection in .NET*. 1. edition. Manning Publications, 2011. ISBN13 978-1935182504.

[9] Nagel, C., Evjen, B., Glynn, J., Watson, K., Skinner, M., *Professional C# 4 and .NET 4*. Wrox, 2010. ISBN 978-0-470-50225-9.

[10] Mackey, A., *Introducing .NET 4.0: With Visual Studio 2010 (Expert's Voice in .NET)*. Apress, 2010. ISBN13 978-1430224556.

[11] Cook, S., Jones, G.,Kent, S.,Wills, A., *Domain Specific Development with Visual Studio DSL Tools*. 1. edition. Addison Wesley, 2007. ISBN 0321398203.

[12] Garofalo, R., *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern*. Microsoft Press, 2011. ISBN 978-0-7356-5092-3.

[13] Hopcroft, J., Motwani, R.,Ullman, J., *Introduction to Automata Theory, Languages, and Computation*. 3. edition. Prentice Hall, 2006. ISBN 9780321455369.

[14] Tomassetti, F., Torchiano, M., *PrEdE: A Projectional Editor for the Eclipse Modeling Framework*. September 2011, Politecnico di Torino, c.so Duca degli Abruzzi, 24, 10129, Torino, Italy

[15] JetBrains, Meta Programming System (MPS), http://www.jetbrains.com/mps

[16] MacDonald, M., *Pro WPF in C# 2010: Windows Presentation Foundation in .NET 4*. 3. edition. Apress, 2010. ISBN13 978-1-4302-7205-2.

[17] Delal, M., Ghoda, A., *XAML Developer Reference*. Microsoft Press, 2011. ISBN13 978-0735658967.

[18] Petzold, C., *Programming Microsoft® Windows® Forms (Pro Developer)*. Microsoft Press, 2005. ISBN13 978-0735621534.

[19] Stephens, R., *WPF Programmer's Reference: Windows Presentation Foundation with C# 2010 and .NET 4*. 1. edition. Wiley Publishing Inc., 2010. ISBN13 978-0470477229.

[20] Sells, C., Griffiths, I., *Programming WPF*. 2. edition. O'REILLY, 2007. ISBN13 978-0596510374.

[21] Microsoft, http://www.microsoft.com

[22] Microsoft, Microsoft Developer Network (MSDN), http://msdn.microsoft.com

[23] Language Workbench Challenge (LWC), http://www.languageworkbenches.net

Content of Attached CD ROM

This thesis is accompanied by the CD ROM. It contains binaries, source code of the prototype implementation and the software prerequisites of the prototype. The CD ROM is organized as follows:

`/README.TXT`
: Brief description of the CD ROM content.

`/doc`
: Electronic version of this thesis.

`/src`
: Source codes of DSLPed.

`/bin/Integration`
: VSIX package containing the DSLPed integration into MSVS.

`bin/AddIns`
: VSIX packages containing prototypes of DSLPed add-ins, especially *Beeps* add-in demonstrated in this thesis.

`/bin/Desktop`
: Desktop integration of DSLPed.

`/bin/Framework`
: DSLPed framework libraries.

`/prerequisites`
: Software prerequisites of DSLPed: *Xsd2Code*.